Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of Booleans
Id	::=	standard identifiers
AExp	::=	*Int*
	|	*Id*
	|	*AExp* `+` *AExp*
	|	*AExp* `/` *AExp*
BExp	::=	*Bool*
	|	*AExp* `<=` *AExp*
	|	`not` *BExp*
	|	*BExp* `and` *BExp*
Stmt	::=	`skip`
	|	*Id* `:=` *AExp*
	|	*Stmt* `;` *Stmt*
	|	`if` *BExp* `then` *Stmt* `else` *Stmt*
	|	`while` *BExp* `do` *Stmt*
Pgm	::=	`var` **List**`{`*Id*`}` `;` *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF.

3.1 IMP: A Simple Imperative Language

To illustrate the various semantic styles discussed in this chapter, we have chosen a small imperative language, called IMP. The IMP language has arithmetic expressions which include the domain of arbitrarily large integer numbers, Boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. All variables used in an IMP program are expected to be declared at the beginning of the program, can only hold integer values (for simplicity, there are no Boolean variables in IMP), and are instantiated with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.5). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

Figure 3.1 shows the syntax of IMP using the algebraic BNF notation. In this book we implicitly assume parentheses as part of any syntax, without defining them explicitly. Parentheses can be freely used for grouping, to increase clarity and/or to avoid ambiguity in parsing. For example, with the syntax in Figure 3.1, $(x + 3) / y$ is a well-formed IMP arithmetic expression.

The only algebraic feature in the IMP syntax in Figure 3.1 is the use of **List**`{`*Id*`}` for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions

Stmt ::= `skip` | *Stmt* `;` *Stmt* with the algebraic production *Stmt* ::= $\mathbf{List}^{\texttt{skip}}_{\texttt{;}}\{\mathit{Stmt}\}$ which captures the idea of statement sequentialization more naturally. Moreover, our syntax for statement sequential composition allows ambiguous parsing. Indeed, if $s_1, s_2, s_3 \in \mathit{Stmt}$ then s_1 `;` s_2 `;` s_3 can be parsed either as (s_1 `;` s_2) `;` s_3 or as s_1 `;` (s_2 `;` s_3). However, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant (but that may not always be the case). It may be worthwhile here pointing out that one should not get tricked by thinking that different parsings mean different evaluation orders. In our case here, both (s_1 `;` s_2) `;` s_3 and s_1 `;` (s_2 `;` s_3) will proceed by evaluating the three statements in order. The difference between the two is that the former will first evaluate s_1 `;` s_2 and then s_3, while the latter will first evaluate s_1 and then s_2 `;` s_3; in either case, s_1, s_2 and s_3 will end up being evaluated in the same order: first s_1, then s_2, and then s_3.

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that `+` is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), `/` is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), `<=` is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that `and` is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

One of the main reasons for which functional language constructs like `+` above are allowed to be non-deterministic in language semantic definitions is because one wants to allow flexibility in how the language is implemented, not because these operations are indeed intended to have fully non-deterministic, or random, behaviors. In other words, their non-determinism is to a large extent an artifact of their intended underspecification. Some language manuals actually state explicitly that one should not rely on the order in which the arguments of language constructs are evaluated. In practice, it is considered to be programmers' responsibility to write their programs in such a way that one does not get different behaviors when the arguments are evaluated in different orders.

To better understand the existing semantic approaches and to expose some of their limitations, Section 3.5 discusses extensions of IMP with expression side effects (a variable increment operation), with abrupt termination (a halt statement), with dynamic threads, with local variable declarations, as well as with all of these together; the resulting language is called IMP++. The extension with side effects, in particular, makes the evaluation strategies of `+`, `<=` and `and` semantically relevant.

Each semantical approach relies on some basic mathematical infrastructure, such as integers, Booleans, etc., because each semantic definition reduces the semantics of the language constructs to those domains. We will assume available any needed mathematical domains, as well as basic operations on them which are clearly tagged (e.g., $+_{Int}$ for addition of integer numbers) to distinguish them from homonymous operations which are language constructs. Unless otherwise stated, we assume no implementation-specific restrictions in our mathematical domains; for example, we assume integer numbers to be arbitrarily large or small. We can think of the underlying domains used in language semantics as parameters of the semantics; indeed, changing the meaning of these domains changes the meaning of all language semantics using them. We also assume that each mathematical domain is endowed with a special element, written $\bot$ for all domains to avoid notational clutter, corresponding to *undefined* values of that domain. Some of these mathematical domains are defined in Chapter 2; appropriate references will be given when such domains are used.

We take the freedom to tacitly use the following naming conventions for IMP-specific terms and/or variables throughout the remainder of this chapter: $x, X \in \mathit{Id}$; $a, A \in \mathit{AExp}$; $b, B \in \mathit{BExp}$; $s, S \in \mathit{Stmt}$; $i, I \in \mathit{Int}$; $t, T \in \mathit{Bool}$; $p, P \in \mathit{Pgm}$. Any of these can be primed or indexed.

sorts:

$Int,\ Bool,\ Id,\ AExp,\ BExp,\ Stmt,\ Pgm$

subsorts:

$Int,\ Id < AExp$

$Bool < BExp$

operations:

$$\begin{array}{rcl}
\texttt{_+_} & : & AExp \times AExp \to AExp \\
\texttt{_/_} & : & AExp \times AExp \to AExp \\
\texttt{_<=_} & : & AExp \times AExp \to BExp \\
\texttt{not_} & : & BExp \to BExp \\
\texttt{_and_} & : & BExp \times BExp \to BExp \\
\texttt{skip} & : & \to Stmt \\
\texttt{_:=_} & : & Id \times AExp \to Stmt \\
\texttt{_;_} & : & Stmt \times Stmt \to Stmt \\
\texttt{if_then_else_} & : & BExp \times Stmt \times Stmt \to Stmt \\
\texttt{while_do_} & : & BExp \times Stmt \to Stmt \\
\texttt{var_;_} & : & \mathbf{List}\{Id\} \times Stmt \to Pgm
\end{array}$$

Figure 3.2: Syntax of IMP as an algebraic signature.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.5, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

☆ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs.

The module `IMP-SYNTAX` in Figure 3.3 imports three builtin modules, namely: `PL-INT`, which we assume it provides a sort `Int`; `PL-BOOL`, which we assume provides a sort `Bool`; and `PL-ID` which we assume provides a sort `Id`. We do not give the precise definitions of these modules here, particularly because one may have many different ways to do it. In our examples from here on in the rest of the chapter we assume that `PL-INT` contains all the integer numbers as constants of sort `Int`, that `PL-BOOL` contains the constants `true` and `false` of sort `Bool`, and that `PL-ID` contains all the letters in the alphabet as constants of sort `Id`. Also, we assume that the module `PL-INT` comes equipped with as many builtin operations on integers as needed.

To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader *not* to use the Maude builtin `INT` and `BOOL` modules, but instead to overwrite them.

```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op not_ : BExp -> BExp [prec 53 format (b o d)] .
  op _and_ : BExp BExp -> BExp [prec 55 format (d b o d)] .
--- Stmt
  sort Stmt .
  op skip : -> Stmt [format (b o)] .
  op _:=_ : Id AExp -> Stmt [prec 40 format (d b o d)] .
  op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d b noi d)] .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 59 format (b o bni n++i bn--i n++i --)] .
  op while_do_ : BExp Stmt -> Stmt [prec 59 format (b o d n++i --)] .
--- Pgm
  sort Pgm .
  op var_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules `PL-INT`, `PL-BOOL` and `PL-ID` defining corresponding sorts `Int`, `Bool`, and `Id`, respectively.

Appendix A.1 shows one possible way to do this in Maude 2.5: we define new modules `PL-INT` and `PL-BOOL` "hooked" to the builtin integer and Boolean values but defining only a subset of operations on them and with names clearly tagged as discussed above, e.g., `_+Int_ : Int Int -> Int`, etc.

Recall from Sections 2.4 and 2.8 that lists, sets, bags, and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort `List{Id}` in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
     var n, s ;
     n := 100 ;
     s := 0 ;
     while not(n <= 0) do (
       s := s + n ;
       n := n + -1
     )
.
```

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". The program `sumPgm` calculates the sum of numbers from 1 to $n = 100$; since we do not have subtraction in IMP, we decremented the value of n by adding -1.

The program `collatzPgm` in Figure 3.4 tests Collatz' conjecture for all numbers n from 1 to $m = 10$, counting the total number of steps in s. The Collatz conjecture, still unsolved, is named after Lothar Collatz (but also known as the $3n + 1$ conjecture), who first proposed it in 1937. Take any natural number n. If n is even, divide it by 2 to get $n/2$, if n is odd multiply it by 3 and

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
eq sumPgm = (
   var n, s ;
   n := 100 ;
   while not(n <= 0) do (
     s := s + n ;
     n := n + -1
   )     ) .

eq collatzStmt = (
   while not (n <= 1) do (
     s := s + 1 ;  q := n / 2 ;  r := q + q + 1 ;
     if r <= n then n := n + n + n + 1 else n := q
   )     ) .

eq collatzPgm = (
   var m, n, q, r, s ;
   m := 10 ;
   while not (m <= 2) do (
     n := m ;
     m := m + -1 ;
     collatzStmt
   )     ) .

eq multiplicationStmt = (        --- fast multiplication (base 2) algorithm
   z := 0 ;
   while not(x <= 0) do (
     q := x / 2 ;
     r := q + q + 1 ;
     if r <= x then z := z + y else skip ;
     x := q ;
     y := y + y
   )     ) .

eq primalityStmt = (
   i := 2 ;  q := n / i ;  t := 1 ;
   while (i <= q and 1 <= t) do (
     x := i ;
     y := q ;
     multiplicationStmt ;
     if n <= z then t := 0 else (i := i + 1 ;  q := n / i)
   )     ) .

eq countPrimesPgm = (
   var i, m, n, q, r, s, t, x, y, z ;
   m := 10 ; n := 2 ;
   while n <= m do (
     primalityStmt ;
     if 1  <= t then s := s + 1 else skip ;
     n := n + 1
   )     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module `IMP-PROGRAMS`.

add 1 to obtain $3n+1$. Repeat the process indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1. Paul Erdös said about the Collatz conjecture: "Mathematics is not yet ready for such problems." While we do not attempt to solve it, we can test it even in a simple language like IMP. It is a good example program to test IMP semantics because it makes use of almost all IMP's language constructs and also has nested loops. The macro `collatzStmt` detaches the check of a single n from the top-level loop iterating n through all $2 < n \leq m$. Note that, since we do not have multiplication and test for even numbers in IMP, we mimic them using the existing IMP constructs.

Finally, the program `countPrimesPgm` counts all the prime numbers up to m. It makes use of `primalityStmt`, which checks whether n is prime or not (writing t to 1 or to 0, respectively), and `primalityStmt` makes use of `multiplicationStmt`, which implements a fast base 2 multiplication algorithm. Defining such a module with programs helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to integer values and stay undefined in the variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[Id \rightharpoonup Int]^{finite}$ (see Section 2.1.2) or, equivalently, to structures of sort $\mathbf{Map}\{Id \mapsto Int\}$ defined using equations (see Section 2.3.2 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a semantic point of view, the equations defining such map structures are invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language (IMP in this case).

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : Id \rightharpoonup Int$, we have a very natural notion of undefinedness for a variable that has not been declared and thus initialized in a state: state σ is considered *undefined* in a variable x if and only if $x \notin Dom(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : State \times Id \rightharpoonup Int$, the terminology *state update* for the operation $\_[\_/\_] : State \times Int \times Id \to State$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{Id\} \times Int \to State$.

Recall from Section 2.1.2 that the lookup operation is itself a partial function, because the state to lookup may be undefined in the variable of interest; as usual, we let $\bot$ denote the undefined state and we write as expected $\sigma(x) = \bot$ and $\sigma(x) \neq \bot$ when the state σ is undefined and, respectively, defined in variable x. Recall also from Section 2.1.2 that the update operation can be used not only to update maps but also to "undefine" them in particular elements: $\sigma[\bot/x]$ is the same as σ in all elements different from x and is undefined in x. Finally, recall also from Section 2.1.2 that the initialization operation yields a partial function mapping each element in the first list argument to the element given as second argument. These can be easily defined equationally, following the equational approach to partial finite-domain functions in Section 2.3.2.

```
mod STATE is including PL-INT + PL-ID .
  sort State .

  op _|->_ : List{Id} Int -> State [prec 0] .
  op .State : -> State .
  op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .

  op _(_) : State Id -> Int [prec 0] .          --- lookup
  op _[_/_] : State Int Id -> State [prec 0] .  --- update

  var Sigma : State .  var I I' : Int .  var X X' : Id .  var Xl : List{Id} .

  eq X |-> undefined = .State .  --- "undefine" a state in a variable

  eq (Sigma & X |-> I)(X) = I .
  eq Sigma(X) = undefined [owise] .

  eq (Sigma &  X |-> I)[I' / X ] = (Sigma & X |-> I') .
  eq Sigma[I / X] = (Sigma & X |-> I) [owise] .

  eq (X,X',Xl) |-> I = X |-> I & X' |-> I & Xl |-> I .
  eq .List{Id} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude.

☆ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Figure 2.2 for our purpose here: the generic sorts `A` (for the source) and `B` (for the target) are replaced by `Id` and `Int`, respectively. Recall from Section 2.3.2 that the constant `undefined` has sort `Undefined`, which is a subsort of all sorts corresponding to mathematical domains (e.g., `Int`, `Bool`, etc.). This way, states can be made "undefined" in certain identifiers by simply updating them in those identifiers with `undefined` (see the equation dissolving undefined bindings in Figure 3.5).

To avoid overloading the comma "`,`" construct for too many purposes (which particularly may confuse Maude's parser), we took the freedom to rename the associative and commutative construct for states to `&`. The only reason for which we bother to give this obvious module is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.1.3 Notes

The style that we follow in this chapter, namely to pick a simple language and then demonstrate the various language definitional approaches by means of that simple language, is quite common. In fact, we named our language IMP after a similar language introduced by Winskel in his book [98], also called IMP, which is essentially identical to ours except that it does not have variable declarations. Since most imperative languages do have variable declarations, we feel it is instructive to include them in our simple language. Winskell gives his IMP a big-step SOS, a small-step SOS, a denotational semantics, and an axiomatic semantics. Later, Nipkow [64] formalized all these

semantics of IMP in the Isabelle/HOL proof assistant [65], and used it to formally relate the various semantics, effectively mechanizing most of Winskel's paper proofs; in doing so, Nipkow [64] found several minor errors in Winskel's proofs, thus showing the benefits of mechanization.

Vardejo and Martì-Oliet [94, 95] show how to use Maude to implement executable semantics for several languages following both big-step and small-step SOS approaches. Like us, they also demonstrate how to define different semantics for the same simple language using different styles; they do so both for an imperative language (very similar to our IMP) and for a functional language. Șerbănuță *et al.* [85] use a similar simple imperative language to also demonstrate how to use rewriting logic to define executable semantics. In fact, this chapter is an extension of [85], both in breadth and in depth. For example, we state and prove general faithful rewriting logic representation results for each of the semantic approaches, while [85] did the same only for the particular simple imperative language considered there. Also, we cover new approaches here, such as denotational semantics, which were not covered in [94, 95, 85].

3.2 Big-Step Structural Operational Semantics (Big-Step SOS)

Known also under the names *natural semantics*, *relational semantics*, and *evaluation semantics*, big-step structural operational semantics, or *big-step SOS* for short, is the most "denotational" of the operational semantics. One can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain. Big-step semantics is so natural, that one is strongly encouraged to use it whenever possible. Unfortunately, as discussed in Section 3.10, big-step semantics has a series of limitations making it inconvenient or impossible to use in many situations, such as when defining control-intensive features or concurrency.

A big-step SOS of a programming language or calculus is given as a formal *proof system* (see Section 2.2). The *big-step SOS sequents* are relations over configurations, typically written $C \Rightarrow R$ or $C \Downarrow R$, with the meaning that R is the configuration obtained after the (complete) evaluation of C. In this book we prefer the notation $C \Downarrow R$. A *big-step SOS rule* therefore has the form

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2 \qquad \ldots \qquad C_n \Downarrow R_n}{C \Downarrow R} \text{ [if } \textit{condition}\text{]}$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, where R, R_1, R_2, ..., R_n are *result configurations*, or *irreducible configurations*, i.e., configurations which cannot be advanced anymore, and where *condition* is an optional *side condition*; as discussed in Section 2.2, the role of side conditions is to filter out undesirable instances of the rule. A big-step semantics compositionally describes how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, etc.). For example, the big-step semantics of IMP addition is

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 +_{Int} i_2\rangle}$$

Here, the meaning of a relation $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that arithmetic expression a is evaluated in state σ to integer i. If expression evaluation has side-effects, then one has to also include a state in the right configurations, so they become of the form $\langle i, \sigma\rangle$ instead of $\langle i\rangle$, as discussed in Section 3.10.

It is common in big-step semantics to not wrap single values in configurations, that is, to write $\langle a, \sigma\rangle \Downarrow i$ instead of $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and similarly for all the other sequents. Also, while the angle-bracket-and-comma notation $\langle \textit{code}, \textit{state}, \ldots\rangle$ is common for configurations, it is not enforced; some prefer to use a square or curly bracket notation of the form $[\textit{code}, \textit{state}, \ldots]$ or $\{\textit{code}, \textit{state}, \ldots\}$, or the simple tuple notation $(\textit{code}, \textit{state}, \ldots)$, or even to use a different (from comma) symbol to separate the various configuration ingredients, e.g., $\langle \textit{code} \mid \textit{state} \mid \ldots\rangle$, etc. Moreover, we may even encounter in the literature sequent notations of the form $\sigma \vdash a \Rightarrow i$ instead of $\langle a, \sigma\rangle \Downarrow \langle i\rangle$, as well as variants of sequent notations that prefer to move various semantic components from the configurations into special, sometimes rather informal, decorations of the symbols $\Downarrow$, $\vdash$ and/or $\Rightarrow$.

For the sake of a uniform notation, in particular when transitioning from languages whose expressions have no side effects to languages whose expressions do have side effects (as we do in Section 3.10), we prefer to always write big-step sequents as $C \Downarrow R$, and always use the angle brackets to surround both configurations involved. This solution is the most general; for example, any additional semantic data or labels that one may need in a big-step definition can be uniformly included as additional components in the configurations (the left ones, or the right ones, or both).

sorts:

Configuration

operations:

$$\begin{array}{rcl} \langle \_, \_ \rangle & : & AExp \times State \to Configuration \\ \langle \_ \rangle & : & Int \to Configuration \\ \langle \_, \_ \rangle & : & BExp \times State \to Configuration \\ \langle \_ \rangle & : & Bool \to Configuration \\ \langle \_, \_ \rangle & : & Stmt \times State \to Configuration \\ \langle \_ \rangle & : & State \to Configuration \\ \langle \_ \rangle & : & Pgm \to Configuration \end{array}$$

Figure 3.6: IMP big-step configurations as an algebraic signature.

3.2.1 IMP Configurations for Big-Step SOS

For the big-step semantics of the simple language IMP, we only need very simple configurations. We follow the comma-and-angle-bracket notational convention, that is, we separate the configuration components by commas and then enclose the entire list with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a Boolean expression b and a state σ. Some configurations may not need a state while others may not need the code. For example, $\langle i \rangle$ is a configuration holding only the integer number i that can be obtained as a result of evaluating an arithmetic expression, while $\langle \sigma \rangle$ is a configuration holding only one state σ that can be obtained after evaluating a statement. Configurations can therefore be of different types and need not necessarily have the same number of components. Here are all the configuration types needed for the big-step semantics of IMP:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions a and states σ;
- $\langle i \rangle$ holding integers i;
- $\langle b, \sigma \rangle$ grouping Boolean expressions b and states σ;
- $\langle t \rangle$ holding truth values $t \in \{true, false\}$;
- $\langle s, \sigma \rangle$ grouping statements s and states σ;
- $\langle \sigma \rangle$ holding states σ;
- $\langle p \rangle$ holding programs p.

IMP Big-Step SOS Configurations as an Algebraic Signature

The configurations above were defined rather informally as tuples of syntax and/or states. There are many ways to rigorously formalize them, all building upon some formal definition of state (besides IMP syntax). Since we have already defined states as partial finite-domain functions (Section 3.1.2) and have already shown how partial finite-domain functions can be formalized as algebraic specifications (Section 2.3.2), we also formalize configurations algebraically.

Figure 3.6 shows an algebraic signature defining the IMP configurations needed for the subsequent big-step operational semantics. For simplicity, we preferred to explicitly define each type of needed

configuration. Consequently, our configurations definition in Figure 3.6 may be more verbose than an alternative polymorphic definition, but we believe that it is clearer for this simple language. We assumed that the sorts *AExp*, *BExp*, *Stmt*, *Pgm* and *State* come from algebraic definitions of the IMP syntax and state, like those in Sections 3.1.1 and 3.1.2; recall that the latter adapted the algebraic definition of partial functions in Section 2.3.2 (see Figure 2.1) as explained in Section 3.1.2.

3.2.2 The Big-Step SOS Rules of IMP

Figure 3.7 shows all the rules in our IMP big-step operational semantics proof system. Recall that the role of a proof system is to prove, or derive, facts. The facts that our proof system will derive have the forms $\langle a, \sigma\rangle \Downarrow \langle i\rangle$, $\langle b, \sigma\rangle \Downarrow \langle t\rangle$, $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$, and $\langle p\rangle \Downarrow \langle \sigma\rangle$ where a ranges over *AExp*, b over *BExp*, s over *Stmt*, p over *Pgm*, i over *Int*, t over *Bool*, and σ and σ' over *State*.

Informally[1], the meaning of derived triples of the form $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that the arithmetic expression a evaluates/executes/transitions to the integer i in state σ; the meaning of $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ is similar but with Boolean values instead of integers. The reason for which it suffices to derive such simple facts is because the evaluation of expressions in our simple IMP language is side-effect-free. When we add the increment operation "`++ x`" in Section 3.10, we will have to change the big-step semantics to work with 4-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$ instead. The meaning of $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ is that the statement s takes state σ to state σ'. Finally, the meaning of pairs $\langle p\rangle \Downarrow \langle \sigma\rangle$ is that the program p yields state σ when executed in the initial state.

In the case of our simple IMP language, the transition relation is going to be *deterministic*, in the sense that $i_1 = i_2$ whenever $\langle a, \sigma\rangle \Downarrow \langle i_1\rangle$ and $\langle a, \sigma\rangle \Downarrow \langle i_2\rangle$ can be deduced (and similarly for Boolean expressions, statements, and programs). However, in the context of non-deterministic languages, triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ state that *a may possibly* evaluate to i in state σ, but it may also evaluate to other integers (and similarly for Boolean expressions, statements, and programs).

The proof system in Figure 3.7 contains one or two rules for each language construct, capturing its intended evaluation relation. Recall from Section 2.2 that proof rules are in fact *rule schemas*, that is, they correspond to (recursively enumerable) sets of *rule instances*, one for each concrete instance of the rule *parameters* (i.e., a, b, σ, etc.). We next discuss each of the rules in Figure 3.7.

The rules (BigStep-Lookup) and (BigStep-Int) define the obvious semantics of variable lookup and integers; these rules have no premises because variables and integers are atomic expressions, so one does not need to evaluate any other subexpression in order to evaluate them. The rule (BigStep-Add) has already been discussed at the beginning of Section 3.2, and (BigStep-Div) is similar. Note that the rules (BigStep-Lookup) and (BigStep-Div) have side conditions. We chose not to short-circuit the division operation when a_1 evaluates to 0. Consequently, no matter whether a_1 evaluates to 0 or not, a_2 is still expected to produce a correct value in order for the rule BigStep-Div to be applicable (e.g., a_2 cannot perform a division by 0).

Before we continue with the remaining rules, let us clarify, using concrete examples, what it means for rule schemas to admit multiple instances and how these can be used to derive proofs. For example, a possible instance of rule (BigStep-Div) can be the following (assume that $x, y \in Id$):

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}$$

[1]Formal definitions of these concepts can only be given after one has a formal language definition. We formally define the notions of evaluation and termination in the context of the IMP language in Definition 16.

$$\langle i, \sigma\rangle \Downarrow \langle i\rangle \qquad \text{(BigStep-Int)}$$

$$\langle x, \sigma\rangle \Downarrow \langle \sigma(x)\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(BigStep-Lookup)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ + } a_2, \sigma\rangle \Downarrow \langle i_1 \ +_{Int} \ i_2\rangle} \qquad \text{(BigStep-Add)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle i_1 \ /_{Int} \ i_2\rangle} \quad \text{if } i_2 \neq 0 \qquad \text{(BigStep-Div)}$$

$$\langle t, \sigma\rangle \Downarrow \langle t\rangle \qquad \text{(BigStep-Bool)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ <= } a_2, \sigma\rangle \Downarrow \langle i_1 \ \leq_{Int} \ i_2\rangle} \qquad \text{(BigStep-Leq)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-Not-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle} \qquad \text{(BigStep-Not-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-And-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle b_2, \sigma\rangle \Downarrow \langle t\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle t\rangle} \qquad \text{(BigStep-And-True)}$$

$$\langle \texttt{skip}, \sigma\rangle \Downarrow \langle \sigma\rangle \qquad \text{(BigStep-Skip)}$$

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle x \texttt{ := } a, \sigma\rangle \Downarrow \langle \sigma[i/x]\rangle} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(BigStep-Asgn)}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle \qquad \langle s_2, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle s_1 \texttt{ ; } s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle} \qquad \text{(BigStep-Seq)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_1\rangle} \qquad \text{(BigStep-If-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle \qquad \langle s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle} \qquad \text{(BigStep-If-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-While-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle s \texttt{;while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-While-True)}$$

$$\frac{\langle s, xl \mapsto 0\rangle \Downarrow \langle \sigma\rangle}{\langle \texttt{ var } xl \texttt{ ; } s\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-Var)}$$

Figure 3.7: BigStep(IMP) — Big-step SOS of IMP ($i, i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List\{}\mathit{Id}\mathbf{\}}$; $a, a_1, a_2 \in \mathit{AExp}$; $t \in \mathit{Bool}$; $b, b_1, b_2 \in \mathit{BExp}$; $s, s_1, s_2 \in \mathit{Stmt}$; $\sigma, \sigma', \sigma_1, \sigma_2 \in \mathit{State}$).

Another instance of rule (BigStep-Div) is the following, which, of course, seems problematic:

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}$$

The rule above is indeed a correct instance of (BigStep-Div), but, however, one will never be able to infer $\langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle$, so this rule can never be applied in a correct inference.

Note, however, that the following is *not* an instance of (BigStep-Div), no matter what ? is chosen to be ($\bot$, or $8/_{Int}0$, etc.):

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 0\rangle}{\langle x/y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle ?\rangle}$$

Indeed, the above does not satisfy the side condition of (BigStep-Div).

The following is a valid proof derivation, where $x, y \in Id$ and $\sigma \in State$ with $\sigma(x) = 8$ and $\sigma(y) = 0$:

$$\dfrac{\dfrac{}{\langle x, \sigma\rangle \Downarrow \langle 8\rangle} \qquad \dfrac{\dfrac{\dfrac{}{\langle y, \sigma\rangle \Downarrow \langle 0\rangle} \qquad \dfrac{}{\langle x, \sigma\rangle \Downarrow \langle 8\rangle}}{\langle y/x, \sigma\rangle \Downarrow \langle 0\rangle} \qquad \dfrac{}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\langle y/x + 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\langle x/(y/x + 2), \sigma\rangle \Downarrow \langle 4\rangle}$$

The proof above can be regarded as a tree, with dots as leaves and instances of rule schemas as nodes. We call such complete (in the sense that their leaves are all dots and their nodes are correct rule instances) trees *proof trees*. This way, we have a way to mathematically *derive facts*, or *sequents*, about programs directly within their semantics. We may call the root of a proof tree the *fact (or sequent) that was proved or derived*, and the tree *its proof or derivation*.

Recall that our original intention was, for demonstration purposes, to attach various evaluation strategies to the arithmetic operations. We wanted `+` and `/` to be non-deterministic and `<=` to be left-right sequential; a non-deterministic evaluation strategy means that the subexpressions are evaluated in any order, possibly interleaving their evaluation steps, which is different from non-deterministically picking an order and then evaluating the subexpressions sequentially in that order. As an analogy, the former corresponds to evaluating the subexpressions concurrently on a multithreaded machine, while the latter to non-deterministically queuing the subexpressions and then evaluating them one by one on a sequential machine. The former has obviously potentially many more possible behaviors than the latter. Note that many programming languages opt for non-deterministic evaluation strategies for their expression constructs precisely to allow compilers to evaluate them in any order or even concurrently; some language manuals explicitly warn the reader not to rely on any evaluation strategy of arithmetic constructs when writing programs.

Unfortunately, big-step semantics is not appropriate for defining non-deterministic evaluation strategies, because such strategies are, by their nature, small-step. One way to do it is to work with sets of values instead of with values and thus associate to each fragment of program in a state the set of all the values that it can non-deterministically evaluate to. However, such an approach would significantly complicate the big-step definition, so we prefer not to do it. Moreover, since IMP has no side effects (until Section 3.10), the non-deterministic evaluation strategies would not lead to non-deterministic results anyway.

We next discuss the big-step rules for Boolean expressions. The rule (BigStep-Bool) is similar to rule (BigStep-Int), but it has only two instances, one for $t = \texttt{true}$ and one for $t = \texttt{false}$. The rules (BigStep-Not-True) and (BigStep-Not-False) are clear; they could have been combined into only one rule if we had assumed our builtin *Bool* equipped with a negation operation. Unlike the division, the conjunction has a short-circuited semantics: if the first conjunct evaluates to `false` then the entire conjunction evaluates to `false` (rule (BigStep-And-False)), and if the first conjunct evaluates to `true` then the conjunction evaluates to whatever truth value the second conjunct evaluates (rule (BigStep-And-True)).

The role of statements in a language is to change the program state. Consequently, the rules for statements derive triples of the form $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ with the meaning that if statement s is executed in state σ and *terminates*, then the resulting state is σ'. We will shortly discuss the aspect of termination in more detail. Rule (BigStep-Skip) states that `skip` does nothing with the state. (BigStep-Asgn) shows how the state σ gets updated by an assignment statement $x := a$ after a is evaluated in state σ using the rules for arithmetic expressions discussed above. (BigStep-Seq) shows how the state updates are propagated by the sequential composition of statements, and rules (BigStep-If-True) and (BigStep-If-False) show how the conditional first evaluates its condition and then, depending upon the truth value of that, it either evaluates its "then" branch or its "else" branch, but never both. The rules giving the big-step semantics of the while loop say that if the condition evaluates to `false` then the while loop dissolves and the state stays unchanged, and if the condition evaluates to `true` then the body followed by the very same while loop is evaluated (rule (BigStep-While-True)). Finally, (BigStep-Var) gives the semantics of programs as the semantics of their statement in a state instantiating all the declared variables to 0.

On Proof Derivations, Evaluation, and Termination

So far we have used the words "evaluation" and "termination" informally. In fact, without a formal definition of a programming language, there is no other way, but informal, to define these notions. Once one has a formal definition of a language, one can not only formally define important concepts like evaluation and termination, but can also rigorously reason about programs. We postpone the subject of program verification until Chapter 14; here we only define and discuss the other concepts.

Definition 16. *Given appropriate* IMP *configurations C and R, the* IMP *big-step sequent $C \Downarrow R$ is **derivable**, written* BigStep(IMP) $\vdash C \Downarrow R$*, iff there is some proof tree rooted in $C \Downarrow R$ which is derivable using the proof system* BigStep(IMP) *in Figure 3.7. Arithmetic (resp. Boolean) expression $a \in AExp$ (resp. $b \in BExp$) **evaluates** to integer $i \in Int$ (resp. to truth value $t \in \{\texttt{true}, \texttt{false}\}$) in state $\sigma \in State$ iff* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i\rangle$ *(resp.* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t\rangle$*). Statement s **terminates** in state σ iff* BigStep(IMP) $\vdash \langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ *for some $\sigma' \in State$; if that is the case, then we say that s **evaluates** in state σ to state σ', or that it **takes** state σ to state σ'. Finally, program p **terminates** iff* BigStep(IMP) $\vdash \langle p\rangle \Downarrow \langle \sigma\rangle$ *for some $\sigma \in State$.*

There are two reasons for which an IMP statement s may not terminate in a state σ: because it may contain a loop that does not terminate, or because it performs a division by zero and thus the rule (BigStep-Div) cannot apply. In the former case, the process of proof search does not terminate, while in the second case the process of proof search terminates in principle, but with a failure to find a proof. Unfortunately, big-step semantics cannot make any distinction between the two reasons for which a proof derivation cannot be found. Hence, the termination notion in Definition 16 rather means *termination with no error*. To catch division-by-zero within the

semantics, we need to add a special *error* value and propagate it through all the language constructs (see Exercise 41).

A formal definition of a language allows to also formally define what it means for the language to be deterministic and to also prove it. For example, we can prove that if an IMP program p terminates then there is a unique state σ such that $\textsc{BigStep}(\text{IMP}) \vdash \langle p \rangle \Downarrow \langle \sigma \rangle$ (see Exercise 42).

Since each rule schema comprises a *recursively enumerable* collection of concrete instance rules, and since our language definition consists of a finite set of rule schemas, by enumerating all the concrete instances of these rules we get a recursively enumerable set of concrete instance rules. Furthermore, since proof trees built with nodes in a recursively enumerable set are themselves recursively enumerable, it follows that the set of proof trees derivable with the proof system in Figure 3.7 is recursively enumerable. In other words, we can find an algorithm that enumerates all the proof trees, in particular one that enumerates all the derivable sequents $C \Downarrow R$. By enumerating all proof trees, given an IMP program p that terminates, one can eventually find the unique state σ such that $\langle p \rangle \Downarrow \langle \sigma \rangle$ is derivable. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is theoretically important to understand that it can be done.

It can be shown that there is no algorithm, based on proof derivation like above or on anything else, which takes as input an IMP program and says whether it terminates or not (see Exercise 43). This follows from the fact that our simple language, due to its while loops and arbitrarily large integers, is Turing-complete. Thus, if one were able to decide termination of programs in our language then one would also be able to decide termination of Turing machines, contradicting one of the basic undecidable problems, the *halting problem* (see Section 4.2 for more on Turing machines).

An interesting observation here is that non-termination of a program corresponds to *lack of proof*, and that the latter is not decidable in many interesting logics. Indeed, in *complete* logics, that is, logics that admit a complete proof system, one can enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, precisely because of the halting problem argument above.

3.2.3 Big-Step SOS in Rewriting Logic

Due to its straightforward recursive nature, big-step semantics is typically easy to represent in other formalisms and also easy to translate into interpreters for the defined languages in any programming language. (The difficulty with big-step semantics is to actually give big-step semantics to complex constructs, as discussed in Section 3.10.) It should therefore come at no surprise to the reader that one can associate a conditional rewrite rule to each big-step rule and hereby obtain a rewriting logic theory that faithfully captures the big-step definition.

In this section we first show that any big-step operational semantics BigStep can be mechanically translated into a rewriting logic theory $\mathcal{R}_{\textsc{BigStep}}$ in such a way that the corresponding derivation relations are step-for-step equivalent, that is, $\textsc{BigStep} \vdash C \Downarrow R$ if and only if $\mathcal{R}_{\textsc{BigStep}} \vdash \mathcal{R}_{C \Downarrow R}$, where $\mathcal{R}_{C \Downarrow R}$ is the corresponding syntactic translation of the big-step sequent $C \Downarrow R$ into a (one-step) rewrite rule. Second, we apply our generic translation technique on the big-step operational semantics $\textsc{BigStep}(\text{IMP})$ and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to the original big-step semantics of IMP. Finally, we show how $\mathcal{R}_{\textsc{BigStep}(\text{Imp})}$ can be seamlessly defined in Maude, thus yielding an interpreter for IMP essentially for free.

Faithful Embedding of Big-Step SOS into Rewriting Logic

To define our translation generically, we need to make some assumptions about the existence of an algebraic axiomatization of configurations. More precisely, as also explained in Section 2.5, we assume that for any parametric term t (can be a configuration, a condition, etc.), the term $\bar{t}$ is an equivalent algebraic variant of t of appropriate sort. For example, by "parametric" configuration we mean a configuration that may possibly make use of parameters, such as $a \in \mathit{AExp}$, $\sigma \in \mathit{State}$, etc.; by "equivalent" algebraic variant we mean a term of sort *Configuration* over an appropriate signature of configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each parameter in C gets replaced by a *variable* of corresponding sort in $\overline{C}$. Similarly, a side condition of a rule can be seen as "parametric", in that it constrains some or all of the parameters involved in the rule; its "equivalent" algebraic variant is an appropriate term of sort *Bool*. Consider, for example, the side condition "$\sigma(x)$ defined" of the rules (BigStep-Lookup) and (BigStep-Asgn) in Figure 3.7; its algebraic variant is the term $\mathit{defined}(\sigma, X)$ of *Bool* sort, where σ and X are variables of sorts *State* and *Id*, respectively (see also Section 3.1.2).

Consider now a general-purpose big-step rule of the form

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2 \qquad \dots \qquad C_n \Downarrow R_n}{C \Downarrow R} \ [\text{if } \mathit{condition}]$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, R, R_1, R_2, ..., R_n are result configurations, and *condition* is some optional side condition. As one may expect, we translate it into the following rewriting logic rule:

$$\overline{C} \to \overline{R} \ \ \textbf{if} \ \ \overline{C_1} \to \overline{R_1} \ \wedge \ \overline{C_2} \to \overline{R_2} \ \wedge \ \dots \ \wedge \ \overline{C_n} \to \overline{R_n} \ [\wedge \ \overline{\mathit{condition}}].$$

Therefore, the big-step SOS rule premises and side conditions are both turned into conditions of the corresponding rewrite rule. The sequent premises become rewrites in the condition, while the side conditions become simple Boolean checks.

We make the reasonable assumption that configurations in BigStep are not nested.

Theorem 2. ***(Faithful embedding of big-step SOS into rewriting logic)*** *For any big-step operational semantics definition* BigStep*, and any* BigStep *appropriate configuration C and result configuration R, the following equivalence holds*

$$\text{BigStep} \ \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \to^1 \overline{R},$$

where $\mathcal{R}_{\text{BigStep}}$ is the rewriting logic semantic definition obtained from BigStep *by translating each rule in* BigStep *as above. (Recall from Section 2.7 that $\to^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewriting logic. Also, as C and R are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{R}$ are ground terms.)*

The non-nestedness assumption on configurations in BigStep guarantees that the resulting rewrite rules in $\mathcal{R}_{\text{BigStep}}$ only apply at the top of the term they rewrite. Since one typically perceives parameters as variables anyway, the only apparent difference between BigStep and $\mathcal{R}_{\text{BigStep}}$ is the different notational conventions they use ($\to$ instead of $\Downarrow$ and conditional rewrite rules instead of conditional deduction rules). As Theorem 2 shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{\text{BigStep}}$ *is* the big-step operational semantics BigStep, and *not* an encoding of it.

At our knowledge, there is no rewrite engine[2] that supports the one-step rewrite relation $\rightarrow^1$ (that appears in Theorem 2). Indeed, rewrite engines aim at high-performance implementations of the general rewrite relation $\rightarrow$, which may even involve parallel rewriting (see Section 2.7 for the precise definitions of $\rightarrow^1$ and $\rightarrow$); $\rightarrow^1$ is meaningful only from a theoretical perspective and there is little to no practical motivation for an efficient implementation of it. Therefore, in order to execute the rewrite theory $\mathcal{R}_{\text{BIGSTEP}}$ resulting from the mechanical translation of big-step semantics BIGSTEP, one needs to take some precautions to ensure that $\rightarrow^1$ is actually identical to $\rightarrow$.

A sufficient condition for $\rightarrow^1$ to be the same as $\rightarrow$ is that the configurations C appearing to the left of $\Downarrow$ are always distinct from those to the right of $\Downarrow$. More generally, if one makes sure that result configurations never appear as left-hand sides of rules in $\mathcal{R}_{\text{BIGSTEP}}$, then one is guaranteed that it is never the case that more than one rewrite step will ever be applied on a given configuration.

Corollary 1. *Under the same hypotheses as in Theorem 2, if result configurations never appear as left-hand sides of rules in* $\mathcal{R}_{\text{BIGSTEP}}$*, then*

$$\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow^1 \overline{R} \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow \overline{R}.$$

Fortunately, in our concrete big-step semantics of IMP, BIGSTEP(IMP), the configurations to the left of $\Downarrow$ and the result configurations to the right of $\Downarrow$ are always distinct. Unfortunately, in general that may not always be the case. For example, when we extend IMP with side effects in Section 3.10, the (possibly affected) state also needs to be part of result configurations, so the semantics of integers is going to be given by an unconditional rule of the form $\langle i, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, which after translation becomes the rewrite rule $\langle i, \sigma \rangle \rightarrow \langle i, \sigma \rangle$. This rule will make the rewrite relation $\rightarrow$ not terminate anymore (although the relation $\rightarrow^1$ terminates). There are at least two simple ways to ensure the hypothesis of Corollary 1:

1. It is highly expected that the only big-step rules in BIGSTEP having a result configuration to the left of $\Downarrow$ are unconditional rules of the form $R \Downarrow R$; such rules typically say that a value is already evaluated. If that is the case, then one can simply drop all the corresponding rules $\overline{R} \rightarrow \overline{R}$ from $\mathcal{R}_{\text{BIGSTEP}}$ and the resulting rewrite theory, say $\mathcal{R}'_{\text{BIGSTEP}}$ still has the property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow \overline{R}$, which is desirable in order to execute the big-step definition on rewrite engines, although the property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow^1 \overline{R}$ will not hold anymore, because, e.g., even though $R \Downarrow R$ is a rule in BIGSTEP, it is not the case that $\mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{R} \rightarrow^1 \overline{R}$.

2. If BIGSTEP contains pairs $R' \Downarrow R$ where R' and R are possibly different result configurations, then one can apply the following general procedure. Change or augment the syntax of the configurations to the left or to the right of $\Downarrow$, so that those changed or augmented configurations will always be different from the other ones. This is the technique employed in our representation of small-step operational semantics in rewriting logic in Section 3.3. More precisely, we prepend all the configurations to the left of the rewrite relation in $\mathcal{R}_{\text{BIGSTEP}}$ with a circle $\circ$, e.g., $\circ C \rightarrow R$, with the intuition that the circled configurations are *active*, while the other ones are *inactive*.

[2] Maude's `rewrite[1]` command does not inhibit the transitive closure of the rewrite relation, it only stops the rewrite engine on a given term after *one* application of a rule on that term; however, many (transitive) applications of rules are allowed when solving the condition of that rule.

$$
\begin{array}{rcl}
\langle I, \sigma\rangle \to \langle I\rangle & & \\
\langle X, \sigma\rangle \to \langle \sigma(X)\rangle & \textbf{if} & \sigma(X) \neq \bot \\
\langle A_1 \texttt{ + } A_2, \sigma\rangle \to \langle I_1 \ +_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle A_1 \texttt{ / } A_2, \sigma\rangle \to \langle I_1 \ /_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \ \wedge\ I_2 \neq 0 \\
\langle A_1 \texttt{ <= } A_2, \sigma\rangle \to \langle I_1 \ \leq_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle T, \sigma\rangle \to \langle T\rangle & & \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{true}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle T\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle B_2, \sigma\rangle \to \langle T\rangle \\
\langle \texttt{skip}, \sigma\rangle \to \langle \sigma\rangle & & \\
\langle X \texttt{ := } A, \sigma\rangle \to \langle \sigma[I/X]\rangle & \textbf{if} & \langle A, \sigma\rangle \to \langle I\rangle \ \wedge\ \sigma(X) \neq \bot \\
\langle S_1 \texttt{;} S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \ \wedge\ \langle S_2, \sigma_1\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_1\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \ \wedge\ \langle S_2, \sigma\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S \texttt{;while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle \\
\langle \texttt{ var } Xl \texttt{ ; } S\rangle \to \langle \sigma\rangle & \textbf{if} & \langle S, Xl \mapsto 0\rangle \to \langle \sigma\rangle
\end{array}
$$

Figure 3.8: $\mathcal{R}_{\text{BIGSTEP(IMP)}}$: the big-step SOS of IMP in rewriting logic.

Regardless of how the desired property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \to \overline{R}$ is ensured, note that, unfortunately, $\mathcal{R}_{\text{BIGSTEP}}$ lacks the main strengths of rewriting logic that make it a good formalism for concurrency: in rewriting logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{BIGSTEP}}$ can only apply at the top, sequentially.

Big-Step SOS of IMP in Rewriting Logic

Figure 3.8 gives the rewriting logic theory $\mathcal{R}_{\text{BIGSTEP(IMP)}}$ that is obtained by applying the procedure above to the big-step semantics of IMP, BIGSTEP(IMP), in Figure 3.7. We have used the rewriting logic convention that variables start with upper-case letters. For the state variable, we used σ, that is, a larger σ symbol. Note how the three side conditions that appear in the proof system in Figure 3.7 turned into normal conditions of rewrite rules. In particular, the two side conditions saying that $\sigma(x)$ is defined became the algebraic term $\sigma(X) \neq \bot$ of Boolean sort.

The following corollary of Theorem 2 and Corollary 1 establishes the faithfulness of the representation of the big-step operational semantics of IMP in rewriting logic:

Corollary 2. $\text{BIGSTEP(IMP)} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP(IMP)}} \vdash \overline{C} \to \overline{R}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system BIGSTEP(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{BIGSTEP(IMP)}}$, so the two are faithfully equivalent.

```
mod IMP-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + STATE .
  sort Configuration .
  op <_,_> : AExp State -> Configuration .
  op   <_> : Int -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op   <_> : Bool -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op   <_> : State -> Configuration .
  op   <_> : Pgm -> Configuration .
endm

mod IMP-SEMANTICS-BIGSTEP is including IMP-CONFIGURATIONS-BIGSTEP .
  var X : Id .  var Xl : List{Id} .  var Sigma Sigma' Sigma1 Sigma2 : State .
  var I I1 I2 : Int .  var T : Bool .
  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .

  rl < I,Sigma > => < I > .
 crl < X,Sigma > => < Sigma(X) >
  if Sigma(X) =/=Bool undefined .
 crl < A1 + A2,Sigma > => < I1 +Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .
 crl < A1 / A2,Sigma > => < I1 /Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > /\ I2 =/=Bool 0 .

  rl < T,Sigma > => < T > .
 crl < A1 <= A2,Sigma > => < I1 <=Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .
 crl < not B,Sigma > => < false >
  if < B,Sigma > => <  true > .
 crl < not B,Sigma > => <  true >
  if < B,Sigma > => < false > .
 crl < B1 and B2,Sigma > => < false >
  if < B1,Sigma > => < false > .
 crl < B1 and B2,Sigma > => < T >
  if < B1,Sigma > => < true > /\ < B2,Sigma > => < T > .

  rl < skip,Sigma > => < Sigma > .
 crl < X := A,Sigma > => < Sigma[I / X] >
  if < A,Sigma > => < I > /\ Sigma(X) =/=Bool undefined .
 crl < S1 ; S2,Sigma > => < Sigma2 >
  if < S1,Sigma > => < Sigma1 > /\ < S2,Sigma1 > => < Sigma2 > .
 crl < if B then S1 else S2,Sigma > => < Sigma1 >
  if < B,Sigma > => < true  > /\ < S1,Sigma > => < Sigma1 > .
 crl < if B then S1 else S2,Sigma > => < Sigma2 >
  if < B,Sigma > => < false > /\ < S2,Sigma > => < Sigma2 > .
 crl < while B do S,Sigma > => < Sigma >
  if < B,Sigma > => < false > .
 crl < while B do S,Sigma > => < Sigma' >
  if < B,Sigma > => < true > /\ < S ; while B do S,Sigma > => < Sigma' > .

 crl < var Xl ; S > => < Sigma >
  if < S,(Xl |-> 0) > => < Sigma > .
endm
```

Figure 3.9: The big-step SOS of IMP in Maude, including the definition of configurations.

☆ Maude Definition of IMP Big-Step SOS

Figure 3.9 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{BIGSTEP(IMP)}}$ in Figure 3.8, including a representation of the algebraic signature in Figure 3.6 for configurations as needed for big-step SOS. The Maude module `IMP-SEMANTICS-BIGSTEP` in Figure 3.9 is executable, so Maude, through its rewriting capabilities, yields an interpreter for IMP; for example, the command

```
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are irrelevant here, so they were replaced by "..."):

```
rewrites: 5118 in ... cpu (... real) (... rewrites/second)
result Configuration: < n |-> 0 , s |-> 5050 >
```

The obtained IMP interpreter actually has acceptable performance; for example, all the programs in Figure 3.4 together take a fraction of a second to execute on conventional PCs or laptops.

In fact, Maude needs only one rewrite logic step to rewrite any configuration; in particular,

```
rewrite [1] < sumPgm > .
```

will give the same output as above. Recall from Section 2.8 that Maude performs a potentially exhaustive search to satisfy the rewrites in rule conditions. Thus, a large number of rule instances can be attempted in order to apply one conditional rule, so a `rewrite [1]` command can take a long time; it may not even terminate. Nevertheless, thanks to Theorem 2, Maude's implicit search mechanism in conditions effectively achieves a proof searcher for big-step SOS derivations.

Once one has a rewriting logic definition in Maude, one can use any of the general-purpose tools provided by Maude on that definition; the rewrite engine is only one of them. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search < sumPgm > =>! Cfg:Configuration .
```

Since our IMP language so far is deterministic, the `search` command will not discover any new behaviors. In fact, the search command will only discover two configurations in total, the original configuration `< sumPgm >` and the result configuration `< n |-> 0 & s |-> 5050 >`. However, as shown in Section 3.5 where we extend IMP with various language features, the `search` command can indeed show all the behaviors of a non-deterministic program (restricted only by the limitations of the particular semantic style employed).

3.2.4 Defining a Type System for IMP Using Big-Step SOS

Big-step SOS is routinely used to define type systems for programming languages, even though in most cases this connection is not made explicit. In this section we demonstrate the use of big-step SOS for defining a type system for IMP, following the same steps as above but more succinctly. Type systems is a broad subject, with many variations and important applications to programming languages. Our intention in this section is twofold: on the one hand we show that big-step SOS is not limited to only defining language semantics, and, on the other hand, we introduce the reader to type systems by means of a very simple example.

The idea underlying big-step SOS definitions of type systems is that a given program or fragment of program in a given type environment reduces, in one big step, to its type. Like states, type

environments are also partial mappings, but from variable names into types instead of values. A common notation for a type judgment is $\Gamma \vdash c : \tau$, where Γ is a type environment, c is a program or fragment, and τ is a type. This type judgment reads "in type environment Γ, program or fragment c has type τ". One can find countless variations of the notation for type judgments in the literature, usually adding more items (pieces of information) to the left of $\vdash$, to its right, or as subscripts or superscripts of it. There is, unfortunately, no well-established notation for all type judgments. Nevertheless, type judgments are special big-step sequents relating two special configurations, one including the givens and the other the results. For example, a simple type judgment $\Gamma \vdash c : \tau$ like above can be regarded as a big-step sequent $\langle c, \Gamma \rangle \Downarrow \langle \tau \rangle$. However, this notation is not preferred.

Figure 3.10 depicts our type system for IMP, which is a nothing but a big-step SOS proof system. We, however, follow the more conventional notation for type judgments discussed above, with one slight change: since in IMP variables are intended to hold only integer values, there is no need for type environments; instead, we replace them by lists of variables, each meant to have the type *int*. Therefore, $xl \vdash c : \tau$ with c and τ as above but with xl a list of variables reads as follows: "when the variables in xl are defined, c has the type τ". We drop the list of variables from the typing judgments of programs, because that would be empty anyway. The big-step SOS rules in Figure 3.10 define the typing policy of each language construct of IMP, guaranteeing all together that a program p types, that is, that $\vdash p : pgm$ is derivable if and only if each construct is used according to its intended typing policy and, moreover, that p declares each variable that is uses. For our simple IMP language, a CFG parser using the syntax defined in Figure 3.1 would already guarantee that each construct is used as intended. Note, however, that the second desired property or our type system (each used variable is declared) is context dependent.

Let us next use the type system in Figure 3.10 to type the program `sumPgm` in Figure 3.4. We split the proof tree in proof subtrees. Note first that using the rules (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Asgn) (second level) and (BigStepTypeSystem-Seq) (third level), we can derive the following proof tree, say $tree_1$:

$$tree_1 \quad = \quad \left\{ \quad \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{100} : int} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{n := 100}) : stmt} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash (\texttt{s := 0}) : stmt}}{\texttt{n,s} \vdash (\texttt{n := 100; s := 0}) : stmt} \right.$$

Similarly, using rules (BigStepTypeSystem-Lookup) and (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Leq) (second level), and (BigStepTypeSystem-Not) (third level), we can derive the following proof tree, say $tree_2$:

$$tree_2 \quad = \quad \left\{ \quad \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : int} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{n <= 0}) : bool}}{\texttt{n,s} \vdash (\texttt{not(n <= 0)}) : bool} \right.$$

$$xl \vdash i : int \qquad \text{(BigStepTypeSystem-Int)}$$

$$(xl, x, xl') \vdash x : int \qquad \text{(BigStepTypeSystem-Lookup)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ + } a_2 : int} \qquad \text{(BigStepTypeSystem-Add)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ / } a_2 : int} \qquad \text{(BigStepTypeSystem-Div)}$$

$$xl \vdash t : bool \qquad \text{(BigStepTypeSystem-Bool)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ <= } a_2 : bool} \qquad \text{(BigStepTypeSystem-Leq)}$$

$$\frac{xl \vdash b : bool}{xl \vdash \texttt{not } b : bool} \qquad \text{(BigStepTypeSystem-Not)}$$

$$\frac{xl \vdash b_1 : bool \qquad xl \vdash b_2 : bool}{xl \vdash b_1 \texttt{ and } b_2 : bool} \qquad \text{(BigStepTypeSystem-And)}$$

$$xl \vdash \texttt{skip} : stmt \qquad \text{(BigStepTypeSystem-Skip)}$$

$$\frac{(xl, x, xl') \vdash a : int}{(xl, x, xl') \vdash (x \texttt{ := } a) : stmt} \qquad \text{(BigStepTypeSystem-Asgn)}$$

$$\frac{xl \vdash s_1 : stmt \qquad xl \vdash s_2 : stmt}{xl \vdash s_1 \texttt{;} s_2 : stmt} \qquad \text{(BigStepTypeSystem-Seq)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s_1 : stmt \qquad xl \vdash s_2 : stmt}{xl \vdash \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 : stmt} \qquad \text{(BigStepTypeSystem-If)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s : stmt}{xl \vdash \texttt{while } b \texttt{ do } s : stmt} \qquad \text{(BigStepTypeSystem-While)}$$

$$\frac{xl \vdash s : stmt}{\vdash \texttt{ var } xl \texttt{ ; } s : pgm} \qquad \text{(BigStepTypeSystem-Var)}$$

Figure 3.10: BigStepTypeSystem(IMP) — Type system of IMP using big-step SOS ($xl, xl' \in \mathbf{List}\{Id\}$; $i \in Int$; $x \in Id$; $a, a_1, a_2 \in AExp$; $t \in Bool$; $b, b_1, b_2 \in BExp$; $s, s_1, s_2 \in Stmt$).

Similarly, we can derive the following proof tree, say $tree_3$:

$$tree_3 \;=\; \left\{ \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{s} : \mathit{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{s + n}) : \mathit{int}} \quad \dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \mathit{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{-1} : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{n + -1}) : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{s := s + n}) : \mathit{stmt}} \quad \dfrac{}{\texttt{n,s} \vdash (\texttt{n := n + -1}) : \mathit{stmt}}}{\texttt{n,s} \vdash (\texttt{s := s + n; n := n + -1}) : \mathit{stmt}} \right.$$

Finally, we can now derive the tree that proves that `sumPgm` is well-typed:

$$\dfrac{\dfrac{tree_1 \qquad \dfrac{tree_2 \qquad tree_3}{\texttt{n,s} \vdash (\texttt{while not(n <= 0) do (s := s + n; n := n + -1)}) : \mathit{stmt}}}{\texttt{n,s} \vdash (\texttt{n := 100; s := 0; while not(n <= 0) do (s := s + n; n := n + -1)}) : \mathit{stmt}}}{\vdash (\texttt{var n,s; n := 100; s := 0; while not(n <= 0) do (s := s + n; n := n + -1)}) : \mathit{pgm}}$$

A major role of a type system is to filter out a set of programs which are obviously wrong. Unfortunately, it is impossible to filter out precisely those programs which would execute erroneously. For example, note that a division is considered type safe whenever its two arguments type to integers, but no check is being made on whether the denominator is 0 or not. Indeed, statically checking whether an expression has a certain value at a certain point in a program is an undecidable problem. Also, no check is being made on whether a detected type error is reachable or not (if unreachable, the detected type error will never show up at runtime). Statically checking whether a certain point in a program is reachable is also an undecidable problem. One should therefore be aware of the fact that in general a type system may allow programs which run into errors when executed and, moreover, that it may reject programs which would execute correctly.

Figure 3.11 shows the straightforward translation of the big-step SOS in Figure 3.10 into a rewriting logic theory, following the general technique described in Section 3.2.3. This translation is based on the obvious reinterpretation of type judgments as big-step SOS sequents mentioned above. The following configurations are used in the rewrite theory in Figure 3.11:

- $\langle a, xl \rangle$ grouping arithmetic expressions a and variable lists xl;
- $\langle b, xl \rangle$ grouping Boolean expressions b and variable lists xl;
- $\langle s, xl \rangle$ grouping statements s and variable lists xl;
- $\langle p \rangle$ holding programs p;
- $\langle \tau \rangle$ holding types τ, which can be *int*, *bool*, *stmt*, or *pgm*.

By Corollary 1 we have that a program p is well-typed, that is, $\vdash p : \mathit{pgm}$ is derivable with the proof system in Figure 3.10, if and only if $\mathcal{R}_{\text{BIGSTEPTYPESYSTEM(IMP)}} \vdash \langle p \rangle \to \langle \mathit{pgm} \rangle$.

☆ Maude Definition of a Type System for IMP using Big-Step SOS

Figure 3.12 shows the Maude representation of the rewrite theory $\mathcal{R}_{\text{BIGSTEPTYPESYSTEM(IMP)}}$ in Figure 3.11, including a representation of the algebraic signature for the needed configurations. The Maude module `IMP-TYPE-SYSTEM-BIGSTEP` in Figure 3.12 is executable, so Maude, through its rewriting capabilities, yields a type checker for IMP; for example, the command

$$
\begin{array}{rl}
\langle I, Xl\rangle \to \langle int\rangle & \\
\langle X, (Xl, X, Xl')\rangle \to \langle int\rangle & \\
\langle A_1 \texttt{ + } A_2, Xl\rangle \to \langle int\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle A_1 \texttt{ / } A_2, Xl\rangle \to \langle int\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle A_1 \texttt{ <= } A_2, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle T, Xl\rangle \to \langle bool\rangle & \\
\langle \texttt{not } B, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \\
\langle B_1 \texttt{ and } B_2, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle B_1, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle B_2, Xl\rangle \to \langle bool\rangle \\
\langle \texttt{skip}, Xl\rangle \to \langle stmt\rangle & \\
\langle X \texttt{ := } A, (Xl, X, Xl')\rangle \to \langle stmt\rangle & \textbf{if } \langle A, (Xl, X, Xl')\rangle \to \langle int\rangle \\
\langle S_1 \texttt{ ; } S_2, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle S_1, Xl\rangle \to \langle stmt\rangle \ \wedge\ \langle S_2, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle S_1, Xl\rangle \to \langle stmt\rangle \ \wedge\ \langle S_2, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{while } B \texttt{ do } S, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle S, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{ var } Xl \texttt{ ; } S\rangle \to \langle pgm\rangle & \textbf{if } \langle S, Xl\rangle \to \langle stmt\rangle
\end{array}
$$

Figure 3.11: $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}}$: the type system of IMP using big-step SOS in rewriting logic.

```
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are irrelevant here, so they were replaced by "..."):

```
rewrites: 19 in ... cpu (... real) (... rewrites/second)
result Configuration: < pgm >
```

A type system is generally expected to be deterministic. Nevertheless, implementations of it (particularly rewrite-based ones) may mistakenly be non-deterministic (non-confluent; see Section 2.6). To gain confidence in the determinism of the Maude definition in Figure 3.12, one may exhaustively search for all possible behaviors yielded by the type checker:

```
search < sumPgm > =>! Cfg:Configuration .
```

As expected, this finds only one solution. This Maude definition of IMP's type checker is very simple and one can easily see that it is confluent (it is orthogonal—see Section 2.6), so the search is redundant. However, the search command may be useful for testing more complex type systems.

3.2.5 Notes

Big-step structural operational semantics (big-step SOS) was introduced under the name *natural semantics* by Kahn [38] in 1987. Even though he introduced it in the limited context of defining Mini-ML, a simple pure (no side effects) version of the ML language, Kahn's aim was to propose natural semantics as a "unified manner to present different aspects of the semantics of programming languages, such as dynamic semantics, static semantics and translation" (Section 1.1 in [38]). Kahn's original notation for big-step sequents was $\sigma \vdash a \Rightarrow i$, with the meaning that a evaluates to i in state (or environment) σ. Kahn, like many others after him (including ourselves; e.g., Section 3.2.4), took the freedom to using a different notation for type judgments, namely $\Gamma \vdash c : \tau$, where Γ is a

```
mod IMP-TYPES is
  sort Type .
  ops int bool stmt pgm : -> Type .
endm

mod IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + IMP-TYPES .
  sort Configuration .
  op <_,_> : AExp List{Id} -> Configuration .
  op <_,_> : BExp List{Id} -> Configuration .
  op <_,_> : Stmt List{Id} -> Configuration .
  op   <_> : Pgm -> Configuration .
  op   <_> : Type -> Configuration .
endm

mod IMP-TYPE-SYSTEM-BIGSTEP is including IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP .
  var X : Id .  var Xl Xl' : List{Id} .  var I : Int .  var T : Bool .
  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .

  rl < I,Xl > => < int > .
  rl < X,(Xl,X,Xl') > => < int > .
 crl < A1 + A2,Xl > => < int >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .
 crl < A1 / A2,Xl > => < int >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .

  rl < T,Xl > => < bool > .
 crl < A1 <= A2,Xl > => < bool >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .
 crl < not B,Xl > => < bool >
  if < B,Xl > => < bool > .
 crl < B1 and B2,Xl > => < bool >
  if < B1,Xl > => < bool > /\ < B2,Xl > => < bool > .

  rl < skip,Xl > => < stmt > .
 crl < X := A,(Xl,X,Xl') > => < stmt >
  if < A,(Xl,X,Xl') > => < int > .
 crl < S1 ; S2,Xl > => < stmt >
  if < S1,Xl > => < stmt > /\ < S2,Xl > => < stmt > .
 crl < if B then S1 else S2,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S1,Xl > => < stmt > /\ < S2,Xl > => < stmt > .
 crl < while B do S,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S,Xl > => < stmt > .

 crl < var Xl ; S > => < pgm >
  if < S,Xl > => < stmt > .
endm
```

Figure 3.12: The type-system of IMP using big-step SOS in Maude, including the definition of types and configurations.

type environment, c is a program or fragment of program, and τ is a type. This colon notation for type judgments was already established in 1987; however, Kahn noticed that the way type systems were defined was a special instance of a more general schema, which he called natural semantics (and which is called big-step SOS here and in many other places). Big-step SOS is very natural when defining pure, sequential and structured languages, so it quickly became very popular. However, Kahn's terminology for "natural" semantics was inspired from its reminiscence to "natural deduction" in mathematical logic, not necessarily from the fact that it is natural to use.

As Kahn himself acknowledged, the idea of using proof systems to capture the operational semantics of programming languages goes back to Plotkin [70, 71] in 1981. Plotkin was the first to coin the terminology *structural operational semantics* (S0S), but what he meant by that was mostly what we call today *small-step* structural operational semantics (small-step SOS). Note, however, that Plotkin in fact used a combination of small-step and big-step SOS, without calling them as such, using the $\rightarrow$ arrow for small-steps and its transitive closure $\rightarrow^*$ for big-steps. We will discuss small-step SOS in depth in Section 3.3. Kahn and others found big-step SOS more natural and convenient than Plotkin's SOS, essentially because it is more abstract and denotational in nature, and one needs fewer rules to define a language semantics.

One of the most notable uses of natural semantics is the formal semantics of Standard ML by Milner *et al.* [58]. Several types of big-step sequents were used in [58], such as $\rho \vdash p \Rightarrow v/f$ for "in environment ρ, sentence p either evaluates to value v or otherwise an error or failure f takes place", and $\sigma, \rho \vdash p \Rightarrow v, \sigma'$ for "in state σ and environment ρ, sentence p evaluates to v and the resulting state is σ'", and $\rho, v \vdash m \Rightarrow v'/f$ for "in environment ρ, a match m either evaluates to v' or otherwise failure f", among many others. After more than twenty years of natural semantics, it is now common wisdom that big-step semantics is inappropriate as a rigorous formalism for defining languages with complex features such as exceptions or concurrency. To give a reasonably compact and readable definition of Standard ML in [58], Milner *et al.* had to make several informal notational conventions, such as a "state convention" to avoid having to mention the state in every rule, and an "exception convention" to avoid having to more than double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [62], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Section 3.5 illustrates in detail the limitations of big-step operational semantics, both with respect to its incapacity of defining certain rather simple language features and with respect to inconvenience in using it (for example, due to its lack of modularity). One of the common uses of natural semantics these days is to define static semantics of programming languages and calculi, such as type systems (see Section 3.2.4).

Hennessy [36] (1990) and Winskel [98] (1993) are perhaps the first textbooks proposing big-step SOS in teaching programming language semantics. They define big-step SOS for several simple languages, including ones similar to the IMP language presented in this chapter. Hennessy [36] defines languages incrementally, starting with a small core and then adding new features one by one, highlighting a major problem with big-step SOS: its lack of modularity. Indeed, the big-step SOS of a language is entirely redefined several times in [36] as new features are added to the language, because adding new features requires changes in the structure of judgments. For example, some big-step SOS judgments in [36] evolve from $a \Rightarrow i$, to $\sigma \vdash a \Rightarrow i$, to $D, \sigma \vdash i$ during the language design experiment, where a is an expression, i an integer, σ a state (or environment), and D a set of function definitions.

While the notations of Hennessy [36] and of Milner *et al.* [58] are somehow reminiscent of original Kahn's notation, Winskel [98] uses a completely different notation. More precisely, he prefers to use

big-step sequents of the form $\langle a, \sigma \rangle \to i$ instead of $\sigma \vdash a \Rightarrow i$. There seems to be, unfortunately, no uniform and/or broadly accepted notation for SOS sequents in the literature, be they big-step or small-step. As already explained earlier in this section, for the sake of uniformity at least throughout this book, we will make and effort to consider sequents of the form $C \Downarrow R$ in our big-step SOS definitions, where C and R are configurations. Similarly, we will make an effort to use the notation $C \to C'$ for small-step sequents (see Section 3.3). We will make it explicit when we deviate from our uniform notation, explaining how the temporary notation relates to the uniform one, as we did in Section 3.2.4.

Big-step SOS is the semantic approach which is probably the easiest to implement in any language or to represent in any computational logic. There are countless approaches to implementing or encoding big-step SOS in various languages or logics, which we cannot enumerate here. We only mention rewriting-based ones which are directly related to the approach followed in this book. Vardejo and Martí-Oliet [95] proposed big-step SOS implementations in Maude for several languages, including Hennessy's languages [36] and Kahn's Mini-ML [38]. Vardejo and Martí-Oliet were mainly interested in demonstrating the strengths of Maude 2 to give executable semantics to concrete languages, rather than in proposing general representations of big-step SOS into rewriting logic that work for any language. In particular, their big-step sequents mimicked those in the original big-step SOS, e.g., they used sequents having the syntax $\sigma \vdash a \Rightarrow i$ in their Maude implementation of Kahn's Mini-ML. Besides Vardejo and Martí-Oliet, several other authors used rewriting logic and Maude to define and implement language semantics for languages or calculi following a small-step approach. These are discussed in Section 3.3.4; we here only want to emphasize that most of those can likely be adapted into big-step SOS definitional or implementation styles, because big-step SOS can be regarded as a special case of small-step SOS, one in which the small step is "big".

3.2.6 Exercises

Prove the following exercises, all referring to the IMP big-step SOS in Figure 3.7.

Exercise 39. *Change the rule* BigStep-Div *so that division short-circuits when a_1 evaluates to 0. (Hint: may need to replace it with two rules, like for the semantics of conjunction).*

Exercise 40. *Change the big-step semantics of the* IMP *conjunction so that it is not short-circuited.*

Exercise 41. *Add an* error *state and modify the big-step semantics in Figure 3.7 to allow derivations of sequents of the form $\langle s, \sigma \rangle \Downarrow \langle \mathsf{error} \rangle$ or $\langle p \rangle \Downarrow \langle \mathsf{error} \rangle$ when s evaluated in state σ or when p evaluated in the initial state performs a division by zero.*

Exercise 42. *Prove that the transition relation defined by the* BigStep(IMP) *proof system in Figure 3.7 is* ***deterministic****, that is:*

- *If* $\text{BigStep(IMP)} \vdash \langle a, \sigma \rangle \Downarrow \langle i \rangle$ *and* $\text{BigStep(IMP)} \vdash \langle a, \sigma \rangle \Downarrow \langle i' \rangle$ *then* $i = i'$*;*
- *If* $\text{BigStep(IMP)} \vdash \langle b, \sigma \rangle \Downarrow \langle t \rangle$ *and* $\text{BigStep(IMP)} \vdash \langle b, \sigma \rangle \Downarrow \langle t' \rangle$ *then* $t = t'$*;*
- *If s terminates in σ then there is a unique σ' such that* $\text{BigStep(IMP)} \vdash \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle$*;*
- *If p terminates then there is a unique σ such that* $\text{BigStep(IMP)} \vdash \langle p \rangle \Downarrow \langle \sigma \rangle$*.*

Prove the same results above for the proof system detecting division-by-zero as in Exercise 41.

Exercise 43. *Show that there is no algorithm, based on the big-step proof system in Figure 3.7 or on anything else, which takes as input an* IMP *program and says whether it terminates* ***or not****.*

3.3 Small-Step Structural Operational Semantics (Small-Step SOS)

Known also under the names *transition semantics*, *reduction semantics*, *one-step operational semantics*, and *computational semantics*, small-step structural operational semantics, or small-step SOS for short, formally captures the intuitive notion of one atomic computational step. Unlike in big-step SOS where one defines all computation steps in one transition, in a small-step SOS definition a transition encodes only one step of computation. To distinguish small-step from big-step transitions, we use a plain arrow $\rightarrow$ instead of $\Downarrow$. To execute a small-step SOS definition, or to relate it to a big-step definition, we need to transitively close the small-step transition relation. Indeed, the conceptual relationship between big-step SOS and small-step SOS is that for any configuration C and any result configuration R, $C \Downarrow R$ if and only if $C \rightarrow^\star R$. Small-step semantics is typically preferred over big-step semantics when defining languages with a high-degree of non-determinism, such as, for example, concurrent languages, because in a small-step semantics one has a direct control over what can execute and when.

Like big-step SOS, a small-step SOS of a programming language or calculus is also given as a formal proof system (see Section 2.2). The *small-step SOS sequents* are also relations of configurations like in big-step SOS, but in small-step SOS they are written $C \rightarrow C'$ and have the meaning that C' is a configuration obtained from C after *one step* of computation. A *small-step SOS rule* therefore has the form

$$\frac{C_1 \rightarrow C_1' \qquad C_2 \rightarrow C_2' \qquad \ldots \qquad C_n \rightarrow C_n'}{C \rightarrow C'} \text{ [if } \textit{condition}\text{]}$$

where C, C', C_1, C_1', C_2, C_2', $\ldots$, C_n, C_n' are configurations holding fragments of program together with all the needed semantic components, like in big-step SOS, and *condition* is an optional side condition. Unlike in big-step SOS, the result configurations do not need to be explicitly defined because in small-step they are implicit: they are precisely those configurations which cannot be reduced anymore using the one-step relation.

Given a configuration holding a fragment of program, a small-step of computation typically takes place in some subpart of the fragment of program. However, when each of the subparts is already reduced, then the small-step can apply on the part itself. A small-step SOS is therefore finer-grain than big-step SOS, and thus more verbose, because one has to cover all the cases where a computation step can take place. For example, the small-step SOS of addition in IMP is

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle}$$

$$\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i_1 +_{Int} i_2, \sigma\rangle$$

Here, the meaning of a relation $\langle a, \sigma\rangle \rightarrow \langle a', \sigma\rangle$ is that arithmetic expression a in state σ is reduced, in one small-step, to arithmetic expression a' and the state stays unchanged. Like for big-step SOS, one can encounter various other notations for small-step SOS configurations in the literature, e.g., $[a, \sigma]$, or (a, σ), or $\{a, \sigma\}$, or $\langle a \mid \sigma\rangle$, etc. Like for big-step SOS, we prefer to uniformly use the angle-bracket-and-comma notation $\langle a, \sigma\rangle$. Also, like for big-step SOS, one can encounter various decorations on the transition arrow $\rightarrow$, a notable situation being when the transition is

labeled. Again like for big-step SOS, we assume that such transition decorations are incorporated in the (source and/or target) configurations. How this can be effectively achieved is discussed in detail in Section 3.6 in the context of modular SOS (which allows rather complex transition labels).

The rules above rely on the fact that expression evaluation in IMP has no side effects. If there were side effects, like in the IMP extension in Section 3.5, then the σ's in the right-hand side configurations above would need to change to a different symbol, say σ', to account for the possibility that the small-step in the condition of the rules, and implicitly in their conclusion, may change the state as well. While in big-step SOS it is more common to derive sequents of the form $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ than of the form $\langle a, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, in small-step SOS the opposite tends to be norm, that is, it is more common to derive sequents of the form $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ than of the form $\langle a, \sigma \rangle \to \langle a' \rangle$. Nevertheless, the latter sequent type also works when defining languages like IMP whose expressions are side-effect-free (see Exercise 52). Some language designers may prefer this latter style, to keep sequents minimal. However, even if one prefers these simpler sequents, we prefer to keep the angle brackets on the right-hand sides of the transition relations (for the same reason like in big-step SOS—to maintain a uniform notation); in other words, we write $\langle a, \sigma \rangle \to \langle a' \rangle$ and not $\langle a, \sigma \rangle \to a'$.

In addition to rules, a small-step SOS may also include *structural identities*. For example, we can state that sequential composition is associative using the following structural identity:

$$(s_1 \; ; \; s_2) \; ; \; s_3 \equiv s_1 \; ; \; (s_2 \; ; \; s_3)$$

The small-step SOS rules apply *modulo* structural identities. In other words, the structural identities can be used anywhere in any configuration and at any moment during the derivation process, without counting as computational steps. In practice, they are typically used to rearrange the syntactic term so that some small-step SOS rule can apply. In particular, the structural rule above allows the designer of the small-step SOS to rely on the fact the first statement in a sequential composition is *not* a sequential composition, which may simplify the actual SOS rules; this is indeed the case in Section 3.5.4, where we extend IMP with dynamic threads (we do not need structural identities in the small-step SOS definition of the simple IMP language in this section). Structural identities are not easy to execute and/or implement in their full generality, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Their role in SOS semantics is the same as the role of equations in rewriting logic definitions; in fact, we effectively turn them into equations when we embed small-step SOS into rewriting logic (see Section 3.3.3).

3.3.1 IMP Configurations for Small-Step SOS

The configurations needed for the small-step semantics of IMP are a subset of those needed for its big-step semantics discussed in Section 3.2.1. Indeed, we still need all the two-component configurations containing a fragment of program and a state, but, for the particular small-step SOS style that we follow in this section, we do not need those result configurations of big-step SOS containing only a value or only a state. If one prefers to instead follow the minimalist style as in Exercise 52, then one would also need the other configuration types. We enumerate all the configuration types needed for the small-step SOS of IMP as given in the reminder of this section:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions a and states σ;
- $\langle b, \sigma \rangle$ grouping Boolean expressions b and states σ;
- $\langle s, \sigma \rangle$ grouping statements s and states σ;

sorts:
Configuration
operations:
$\langle \_, \_ \rangle$: $AExp \times State \to Configuration$
$\langle \_, \_ \rangle$: $BExp \times State \to Configuration$
$\langle \_, \_ \rangle$: $Stmt \times State \to Configuration$
$\langle \_ \rangle$: $Pgm \to Configuration$

Figure 3.13: IMP small-step SOS configurations as an algebraic signature.

- $\langle p \rangle$ holding programs p.

We still need the one-component configuration holding only a program, because we still want to reduce a program in the default initial state (empty) without having to mention the empty state.

IMP Small-Step SOS Configurations as an Algebraic Signature

Figure 3.13 shows an algebraic signature defining the IMP configurations above, which is needed for the subsequent small-step operational semantics. We defined this algebraic signature in the same style and following the same assumptions as those for big-step SOS in Section 3.2.1.

3.3.2 The Small-Step SOS Rules of IMP

Figures 3.14 and 3.15 show all the rules in our IMP small-step SOS proof system, the former showing the rules corresponding to expressions, both arithmetic and Boolean, and the latter showing those rules corresponding to statements. The sequents that this proof system derives have the forms $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$, $\langle b, \sigma \rangle \to \langle b', \sigma \rangle$, $\langle s, \sigma \rangle \to \langle s', \sigma' \rangle$, and $\langle p \rangle \to \langle s, \sigma \rangle$, where a ranges over *AExp*, b over *BExp*, s over *Stmt*, p over *Pgm*, and σ and σ' over *State*.

The meaning of derived triples of the form $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ is that given state σ, the arithmetic expression a reduces in one (small) step to the arithmetic expression a' and the state σ stays unchanged. The meaning of $\langle b, \sigma \rangle \to \langle b', \sigma \rangle$ is similar but with Boolean expressions instead of arithmetic expressions. The meaning of $\langle s, \sigma \rangle \to \langle s', \sigma' \rangle$ is that statement s in state σ reduces in one step to statement s' in a potentially modified state σ'. The meaning of $\langle p \rangle \to \langle s, \sigma \rangle$ is that program p reduces in one step to statement s in state σ (as expected, whenever such sequents can be derived, the statement s is the body of p and the state σ initializes to 0 the variables declared by p). The reason for which the state stays unchanged in the sequents corresponding to arithmetic and Boolean expressions is because, as discussed, IMP's expressions currently have no side effects; we will have to change these rules later on in Section 3.5 when we add a variable increment arithmetic expression construct to IMP. A small-step reduction of a statement may or may not change the state, so we use a different symbol in the right-hand side of statement transitions, σ', to cover both cases.

We next discuss each of the small-step SOS rules of IMP in Figures 3.14 and 3.15. Before we start, note that we have no small-step rules for reducing constant (integer or Boolean) expressions to their corresponding values as we had in big-step SOS (i.e., no rules corresponding to (BigStep-Int) and (BigStep-Bool) in Figure 3.7). Indeed, we do not want to have small-step SOS rules of the form $\langle v, \sigma \rangle \to \langle v, \sigma \rangle$ because no one-step reductions are further desired on values v: adding such rules would lead to undesired divergent SOS reductions later on when we consider the transitive closure

$$\langle x, \sigma\rangle \to \langle \sigma(x), \sigma\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(SmallStep-Lookup)}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \texttt{+} a_2, \sigma\rangle \to \langle a_1' \texttt{+} a_2, \sigma\rangle} \qquad \text{(SmallStep-Add-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \texttt{+} a_2, \sigma\rangle \to \langle a_1 \texttt{+} a_2', \sigma\rangle} \qquad \text{(SmallStep-Add-Arg2)}$$

$$\langle i_1 \texttt{+} i_2, \sigma\rangle \to \langle i_1 +_{Int} i_2, \sigma\rangle \qquad \text{(SmallStep-Add)}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \texttt{/} a_2, \sigma\rangle \to \langle a_1' \texttt{/} a_2, \sigma\rangle} \qquad \text{(SmallStep-Div-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \texttt{/} a_2, \sigma\rangle \to \langle a_1 \texttt{/} a_2', \sigma\rangle} \qquad \text{(SmallStep-Div-Arg2)}$$

$$\langle i_1 \texttt{/} i_2, \sigma\rangle \to \langle i_1 /_{Int} i_2, \sigma\rangle \quad \text{if } i_2 \neq 0 \qquad \text{(SmallStep-Div)}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \texttt{<=} a_2, \sigma\rangle \to \langle a_1' \texttt{<=} a_2, \sigma\rangle} \qquad \text{(SmallStep-Leq-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle i_1 \texttt{<=} a_2, \sigma\rangle \to \langle i_1 \texttt{<=} a_2', \sigma\rangle} \qquad \text{(SmallStep-Leq-Arg2)}$$

$$\langle i_1 \texttt{<=} i_2, \sigma\rangle \to \langle i_1 \leq_{Int} i_2, \sigma\rangle \qquad \text{(SmallStep-Leq)}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{not}\, b, \sigma\rangle \to \langle \texttt{not}\, b', \sigma\rangle} \qquad \text{(SmallStep-Not-Arg)}$$

$$\langle \texttt{not true}, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle \qquad \text{(SmallStep-Not-True)}$$

$$\langle \texttt{not false}, \sigma\rangle \to \langle \texttt{true}, \sigma\rangle \qquad \text{(SmallStep-Not-False)}$$

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \to \langle b_1' \texttt{ and } b_2, \sigma\rangle} \qquad \text{(SmallStep-And-Arg1)}$$

$$\langle \texttt{false and}\, b_2, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle \qquad \text{(SmallStep-And-False)}$$

$$\langle \texttt{true and}\, b_2, \sigma\rangle \to \langle b_2, \sigma\rangle \qquad \text{(SmallStep-And-True)}$$

Figure 3.14: SmallStep(IMP) — Small-step SOS of IMP expressions ($i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $a_1, a_1', a_2, a_2' \in \mathit{AExp}$; $b, b', b_1, b_1', b_2 \in \mathit{BExp}$; $\sigma \in \mathit{State}$).

of the one-step relation $\rightarrow$. Recall that the big-step SOS relation $\Downarrow$ captured all the reduction steps at once, including zero steps, and thus it did not need to be transitively closed like the small-step relation $\rightarrow$, so evaluating values to themselves was not problematic in big-step SOS.

The rule (SmallStep-Lookup) happens to be almost the same as in the big-step SOS; that's because variable lookup is an atomic-step operation both in big-step and in small-step SOS. The rules (SmallStep-Add-Arg1), (SmallStep-Add-Arg2), and (SmallStep-Add) give the small-step semantics of addition, and (SmallStep-Div-Arg1), (SmallStep-Div-Arg2), and (SmallStep-Div) give the small-step semantics of division, each covering all the three cases where a small-step reduction can take place. The first two cases in each group may apply non-deterministically. Recall from Section 3.2 that big-step SOS was inappropriate for defining the desired non-deterministic evaluation strategies for `+` and `/`. Fortunately, that was not a big problem for IMP, because its intended non-deterministic constructs are side-effect free. Therefore, the intended non-deterministic evaluation strategies of these particular language constructs did not affect the overall determinism of the IMP language, thus making its deterministic (see Exercise 42) big-step SOS definition in Section 3.2 acceptable. As expected, the non-deterministic evaluation strategies of `+` and `/`, which this time can be appropriately captured within the small-step SOS, will not affect the overall determinism of the IMP language (that is, the reflexive/transitive closure $\rightarrow^\star$ of $\rightarrow$; see Theorem 3).

The rules (SmallStep-Leq-Arg1), (SmallStep-Leq-Arg2), and (SmallStep-Leq) give the deterministic, sequential small-step SOS of `<=`. The first rule applies whenever a_1 is not an integer, then the second rule applies when a_1 is an integer but a_2 is not an integer, and finally, when both a_1 and a_2 are integers, the third rule applies. The rules (SmallStep-Not-Arg), (SmallStep-Not-True), and (SmallStep-Not-False) are self-explanatory, while the rules (SmallStep-And-Arg1), (SmallStep-And-False) and (SmallStep-And-True) give the short-circuited semantics of `and`: indeed, b_2 will not be reduced unless b_1 is first reduced to `true`.

Before we continue with the remaining small-step SOS rules for statements, let us see an example of a small-step SOS reduction using the rules discussed so far; as in the case of the big-step SOS rules in Section 3.2, recall that the small-step SOS rules are also rule schemas, that is, they are parametric in the (meta-)variables a, a_1, b, s, σ, etc. The following is a correct derivation, where x and y are any variables and σ is any state with $\sigma(x) = 1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle x, \sigma\rangle \rightarrow \langle 1, \sigma\rangle}}{\langle y \,/\, x, \sigma\rangle \rightarrow \langle y \,/\, 1, \sigma\rangle}}{\langle x + (y \,/\, x), \sigma\rangle \rightarrow \langle x + (y \,/\, 1), \sigma\rangle}}{\langle (x + (y \,/\, x)) \mathrel{<=} x, \sigma\rangle \rightarrow \langle (x + (y \,/\, 1)) \mathrel{<=} x, \sigma\rangle}$$

The above can be regarded as a proof of the fact that replacing the second occurrence of x by 1 is a correct one-step computation of IMP, as defined using the small-step SOS rules discussed so far.

Let us now discuss the small-step SOS rules of statements in Figure 3.15. Unlike the reduction of expressions, a reduction step of a statement may also change the state. Rule (SmallStep-Asgn-Arg2) reduces the second argument—which is an arithmetic expression—of an assignment statement whenever possible, regardless of whether the assigned variable was declared or not. Exercise 46 proposes an alternative semantics where the arithmetic expression is only reduced when the assigned variable has been declared. When the second argument is already fully reduced (i.e., it is an integer value), the rule (SmallStep-Asgn) reduces the assignment statement to `skip`, at the same time updating the state accordingly. Therefore, two steps are needed in order to assign an already evaluated expression to a variable: one step to write the variable in the state and modify

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle x := a, \sigma\rangle \to \langle x := a', \sigma\rangle} \qquad \text{(SmallStep-Asgn-Arg2)}$$

$$\langle x := i, \sigma\rangle \to \langle \texttt{skip}, \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(SmallStep-Asgn)}$$

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1 \texttt{ ; } s_2, \sigma\rangle \to \langle s_1' \texttt{ ; } s_2, \sigma'\rangle} \qquad \text{(SmallStep-Seq-Arg1)}$$

$$\langle \texttt{skip ; } s_2, \sigma\rangle \to \langle s_2, \sigma\rangle \qquad \text{(SmallStep-Seq-Skip)}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle \texttt{if } b' \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle} \qquad \text{(SmallStep-If-Arg1)}$$

$$\langle \texttt{if true then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle s_1, \sigma\rangle \qquad \text{(SmallStep-If-True)}$$

$$\langle \texttt{if false then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle s_2, \sigma\rangle \qquad \text{(SmallStep-If-False)}$$

$$\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \to \langle \texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else skip}, \sigma\rangle \qquad \text{(SmallStep-While)}$$

$$\langle \texttt{var } xl \texttt{ ; } s\rangle \to \langle s, (xl \mapsto 0)\rangle \qquad \text{(SmallStep-Var)}$$

Figure 3.15: SmallStep(IMP) — Small-step SOS of IMP statements ($i \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List}\{\mathit{Id}\}$; $a, a' \in \mathit{AExp}$; $b, b' \in \mathit{BExp}$; $s, s_1, s_1', s_2 \in \mathit{Stmt}$; $\sigma, \sigma' \in \mathit{State}$).

$$C \to^* C \qquad \text{(SmallStep-Closure-Stop)}$$

$$\frac{C \to C'', \quad C'' \to^* C'}{C \to^* C'} \qquad \text{(SmallStep-Closure-More)}$$

Figure 3.16: SmallStep(IMP) — Reflexive/transitive closure of the small-step SOS relation, which is the same for any small-step SOS of any language or calculus ($C, C', C'' \in \mathit{Configuration}$).

the assignment to `skip`, and another step to dissolve the resulting `skip`. Exercise 47 proposes an alternative semantics where these operations can be done in one step.

The rules (SmallStep-Seq-Arg1) and (SmallStep-Seq-Skip) give the small-step SOS of sequential composition: if the first statement is reducible then reduce it, otherwise, if it is `skip`, move on in a small-step to the second statement. Another possibility (different from that in Exercise 47) to avoid wasting the computational step generated by reductions to `skip` like in the paragraph above, is to eliminate the rule (SmallStep-Seq-Skip) and instead to add a rule that allows the reduction of the second statement provided that the first one is terminated. This approach is proposed by Hennessy [36], where he introduces a new sequent for *terminated configurations*, say $C\surd$, and then includes a rule like the following (and no rule like our (SmallStep-Seq-Skip)):

$$\frac{\langle s_1, \sigma\rangle\surd \qquad \langle s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}{\langle s_1 \texttt{ ; } s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}$$

The three rules for the conditional, namely (SmallStep-If-Arg1), (SmallStep-If-True), and (SmallStep-If-False), are straightforward; note that the two branches are never reduced when the condition can still be reduced. Exercise 49 proposes an alternative semantics for the conditional which wastes no computational step on switching to one of the two branches once the condition is evaluated.

The small-step SOS of `while` unrolls the loop once; this unrolling semantics seems as natural as it can be, but one should notice that it actually also generates an artificial computational step. Exercise 50 proposes an alternative semantics for `while` which wastes no computational step.

Finally, (SmallStep-Var) gives the semantics of programs by reducing them to their body statement in the expected state formed by initializing all the declared variables to 0. Note, however, that this rule also wastes a computational step; indeed, one may not want the initialization of the state with default values for variables to count as a step. Exercise 51 addresses this problem.

It is worthwhile noting that one has some flexibility in how to give a small-step SOS semantics to a language. The same holds true for almost any language definitional style, not only for SOS.

On Proof Derivations, Evaluation, and Termination

To formally capture the notion of "sequence of transitions", in Figure 3.16 we define the relation of *reflexive/transitive closure* of the small-step SOS transition.

Definition 17. *Given appropriate* IMP *small-step SOS configurations C and C', the* IMP *small-step SOS sequent $C \to C'$ is* ***derivable****, written* SmallStep(IMP) $\vdash C \to C'$*, iff there is some proof tree rooted in $C \to C'$ which is derivable using the proof system* SmallStep(IMP) *in Figures 3.14 and 3.15. In this case, we also say that C* ***reduces in one step*** *to C'. Similarly, the many-step*

sequent $C \rightarrow^\star C'$ *is* ***derivable****, written* SmallStep(IMP) $\vdash C \rightarrow^\star C'$*, iff there is some proof tree rooted in* $C \rightarrow^\star C'$ *which is derivable using the proof system in Figures 3.14, 3.15, and 3.16. In this case, we also say that* C ***reduces*** *(in zero, one, or more steps) to* C'*. Configuration* R *is* ***irreducible*** *iff there is no configuration* C *such that* SmallStep(IMP) $\vdash R \rightarrow C$*, and is a* ***result*** *iff it has one of the forms* $\langle i, \sigma \rangle$*,* $\langle t, \sigma \rangle$*, or* $\langle \texttt{skip}, \sigma \rangle$*, where* $i \in Int$*,* $t \in Bool$*, and* $\sigma \in State$*. Finally, configuration* C ***terminates*** *under* SmallStep(IMP) *iff there is no infinite sequence of configurations* $C_0, C_1, \ldots$ *such that* $C_0 = C$ *and* C_i *reduces in one step to* C_{i+1} *for any natural number* i*.*

Result configurations are irreducible, but there are irreducible configurations which are not necessarily result configurations. For example, the configuration $\langle i \,/\, 0, \sigma \rangle$ is irreducible but it is not a result. Like for big-step SOS, to catch division-by-zero within the semantics we need to add special error values/states and propagate them through all the language constructs (see Exercise 54).

The syntax of IMP (Section 3.1.1, Figure 3.1) was deliberately ambiguous with regards to sequential composition, and that was motivated by the fact that the semantics of the language will be given in such a way that the syntactic ambiguity will become irrelevant. We can now rigorously prove that is indeed the case, that is, we can prove properties of the like "SmallStep(IMP) $\vdash \langle (s_1 \,;\, s_2) \,;\, s_3, \sigma \rangle \rightarrow \langle (s_1' \,;\, s_2) \,;\, s_3, \sigma' \rangle$ if and only if SmallStep(IMP) $\vdash \langle s_1 \,;\, (s_2 \,;\, s_3), \sigma \rangle \rightarrow \langle s_1' \,;\, (s_2 \,;\, s_3), \sigma' \rangle$", etc. Exercise 55 discusses several such properties which, together with the fact that the semantics of no language construct is structurally defined in terms of sequential composition, also says that adding the associativity of sequential composition as a structural identity to the small-step SOS of IMP does not change the set of global behaviors of any IMP program (though we do not add it). However, that will not be the case anymore when we extend IMP with dynamic threads in Section 3.5.4, because the semantics of thread spawning will be given making use of sequential composition in such a way that adding this structural identity will be necessary in order to capture the desired set of behaviors.

Note that there are non-terminating sequences which repeat configurations, as well as non-terminating sequences in which all the configurations are distinct; an example of the former is a sequence generated by reducing the statement "`while true do skip`", while an example of the latter is a sequence generated by reducing "`while (n > 0) do n := n + 1`". Nevertheless, in the case of IMP, a configuration terminates if and only if it reduces to some irreducible configuration (see Exercise 56). This is not necessarily the case for non-deterministic languages, such as the IMP++ extension in Section 3.5, because reductions of configurations in such language semantics may non-deterministically choose steps that lead to termination, as well as steps that may not lead to termination. In the case of IMP though, the local non-determinism given by rules like (SmallStep-Add-Arg1) and (SmallStep-Add-Arg2) does not affect the overall determinism of the IMP language (Exercise 57).

Relating Big-Step and Small-Step SOS

As expected, the reflexive/transitive closure $\rightarrow^\star$ of the small-step SOS relation captures the same complete evaluation meaning as the $\Downarrow$ relation in big-step SOS (Section 3.2). Since for demonstrations reasons we deliberately worked with different result configurations in big-step and in small-step SOS, our theorem below looks slightly more involved than usual; if we had the same configurations in both semantics, then the theorem below would simply state "for any configuration C and any result configuration R, SmallStep(IMP) $\vdash C \rightarrow^\star R$ if and only if BigStep(IMP) $\vdash C \Downarrow R$":

Theorem 3. *The following equivalences hold for any $a \in AExp$, $i \in Int$, $b \in BExp$, $t \in Bool$, $s \in Stmt$, $p \in Pgm$, and $\sigma, \sigma' \in State$:*

- $\textsc{SmallStep}(\text{IMP}) \vdash \langle a, \sigma \rangle \rightarrow^\star \langle i, \sigma' \rangle$ *for a state* σ' *iff* $\textsc{BigStep}(\text{IMP}) \vdash \langle a, \sigma \rangle \Downarrow \langle i \rangle$;
- $\textsc{SmallStep}(\text{IMP}) \vdash \langle b, \sigma \rangle \rightarrow^\star \langle t, \sigma' \rangle$ *for a state* σ' *iff* $\textsc{BigStep}(\text{IMP}) \vdash \langle b, \sigma \rangle \Downarrow \langle t \rangle$;
- $\textsc{SmallStep}(\text{IMP}) \vdash \langle s, \sigma \rangle \rightarrow^\star \langle\, \texttt{skip}\, , \sigma' \rangle$ *for a state* σ' *iff* $\textsc{BigStep}(\text{IMP}) \vdash \langle s, \sigma \rangle \Downarrow \langle\, \texttt{skip}\, \rangle$;
- $\textsc{SmallStep}(\text{IMP}) \vdash \langle p \rangle \rightarrow^\star \langle\, \texttt{skip}\, , \sigma \rangle$ *for a state* σ *iff* $\textsc{BigStep}(\text{IMP}) \vdash \langle p \rangle \Downarrow \langle \sigma \rangle$.

Note that the small-step SOS relation for IMP is a *recursive*, or *decidable* problem: indeed, given configurations C and C', one can use the small-step proof system in Figures 3.14 and 3.15 to exhaustively check whether indeed $C \rightarrow C'$ is derivable or not. Moreover, since the rules for the reflexive/transitive closure relation $\rightarrow^\star$ in Figure 3.16 can be used to systematically generate any sequence of reductions, we conclude that the relation $\rightarrow^\star$ is *recursively enumerable*. Theorem 3 together with the discussion at the end of Section 3.2.2 and Exercise 43, tell us that $\rightarrow^\star$ is properly recursively enumerable, that is, it cannot be recursive. This tells us, in particular, that non-termination of a program p is equivalent to saying that, no matter what the state σ is, $\textsc{SmallStep}(\text{IMP}) \vdash \langle p \rangle \rightarrow^\star \langle\, \texttt{skip}\, , \sigma \rangle$ cannot be derived. However, unlike for big-step SOS where nothing else can be said about non-terminating programs, in the case of small-step SOS definitions one can use the small-step relation, $\rightarrow$, to observe program executions for any number of steps.

3.3.3 Small-Step SOS in Rewriting Logic

Like for big-step SOS, we can also associate a conditional rewrite rule to each small-step SOS rule and hereby obtain a rewriting logic theory that faithfully (i.e., step-for-step) captures the small-step SOS definition. Additionally, we can associate a rewriting logic equation to each SOS structural identity, because in both cases rules are applied modulo structural identities or equations. An important technical aspect needs to be resolved, though. The rewriting relation of rewriting logic is by its own nature reflexively and transitively closed. On the other hand, the small-step SOS relation is not reflexively and transitively closed by default (its reflexive/transitive closure is typically defined a posteriori, as we did in Figure 3.13). Therefore, we need to devise mechanisms to inhibit rewriting logic's reflexive, transitive and uncontrolled application of rules.

We first show that any small-step SOS, say SmallStep, can be mechanically translated into a rewriting logic theory, say $\mathcal{R}_{\textsc{SmallStep}}$, in such a way that the corresponding derivation relations are step-for-step equivalent, that is, $\textsc{SmallStep} \vdash C \rightarrow C'$ if and only if $\mathcal{R}_{\textsc{SmallStep}} \vdash \mathcal{R}_{C \rightarrow C'}$, where $\mathcal{R}_{C \rightarrow C'}$ is the corresponding syntactic translation of the small-step SOS sequent $C \rightarrow C'$ into a rewriting logic sequent. Second, we apply our generic translation technique on the small-step SOS formal system $\textsc{SmallStep}(\text{IMP})$ defined in Section 3.3.2 and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to the original small-step SOS of IMP. Finally, we show how $\mathcal{R}_{\textsc{SmallStep}(\text{IMP})}$ can be seamlessly defined in Maude, thus yielding another interpreter for IMP (in addition to the one similarly obtained from the big-step SOS of IMP in Section 3.2.3).

Faithful Embedding of Small-Step SOS into Rewriting Logic

Like for big-step SOS (Section 3.2.3), to define our translation from small-step SOS to rewriting logic generically, we assume that each parametric configuration C admits an equivalent algebraic

variant $\overline{C}$ as a term of sort *Configuration* over an appropriate signature of configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each *parameter* in C (e.g., arithmetic expression $a \in AExp$) gets replaced by a *variable* of corresponding sort in $\overline{C}$ (e.g., variable A of sort $AExp$). Consider now a general-purpose small-step SOS rule of the form

$$\frac{C_1 \to C_1' \qquad C_2 \to C_2' \qquad \dots \qquad C_n \to C_n'}{C \to C'} \ [\text{if } \textit{condition}]$$

where C, C', C_1, C_1', C_2, C_2', ..., C_n, C_n' are configurations holding fragments of program together with all the needed semantic components, and *condition* is an optional side condition. Before we introduce our transformation, let us first discuss why the same straightforward transformation that we used in the case of big-step SOS,

$$\overline{C} \to \overline{C'} \ \textbf{ if } \ \overline{C_1} \to \overline{C_1'} \ \wedge \ \overline{C_2} \to \overline{C_2'} \ \wedge \ \dots \ \wedge \ \overline{C_n} \to \overline{C_n'} \ [\wedge \ \overline{\textit{condition}}],$$

does not work in the case of small-step SOS. For example, with that transformation, the rewrite rules corresponding to the small-step SOS rules of IMP for assignment (SmallStep-Asgn-Arg2) and (SmallStep-Asgn) in Figure 3.15 would be

$$\begin{array}{rll} \langle X := A, \sigma \rangle \to \langle X := A', \sigma \rangle & \textbf{if} & \langle A, \sigma \rangle \to \langle A', \sigma \rangle \\ \langle X := I, \sigma \rangle \to \langle \, \texttt{skip} \, , \sigma[I/X] \rangle & \textbf{if} & \sigma(X) \neq \bot \end{array}$$

The problem with these rules is that the rewrite of a configuration of the form $\langle x := i, \sigma \rangle$ for some $x \in Id$, $i \in Int$ and $\sigma \in State$ may not terminate, applying forever the first rule: in rewriting logic, $\langle i, \sigma \rangle \to \langle i, \sigma \rangle$ because $\to$ is closed under reflexivity. Even if we may somehow solve this reflexivity aspect by defining and then including an additional condition $A \neq A'$, such rules still fail to capture the intended small-step transition, because $\to$ is also closed transitively in rewriting logic, so there could be many small-steps taking place in the condition of the first rule before the rule is applied.

To capture *exactly one step* of reduction, thus avoiding the inherent automatic reflexive and transitive closure of the rewrite relation which is desirable in rewriting logic but not in reduction semantics, we can mark the left-hand side (or, alternatively, the right-hand side) configuration in each rewrite sequent to always be distinct from the other one; then each rewrite sequent comprises precisely one step, from a marked to an unmarked configuration (or vice versa). For example, let us place a $\circ$ in front of all the left-hand side configurations and keep the right-hand side configurations unchanged. Then the generic small-step SOS rule above translates into the rewriting logic rule

$$\circ\overline{C} \to \overline{C'} \ \textbf{ if } \ \circ\overline{C_1} \to \overline{C_1'} \ \wedge \ \circ\overline{C_2} \to \overline{C_2'} \ \wedge \ \dots \ \wedge \ \circ\overline{C_n} \to \overline{C_n'} \ [\wedge \ \overline{\textit{condition}}].$$

One can metaphorically think of a marked configuration $\circ\overline{C}$ as a hot configuration that needs to be cooled down in one step, while of an unmarked configuration $\overline{C}$ as a cool one. Theorem 4 below shows as expected that a small-step SOS sequent $C \to C'$ is derivable if and only if the term $\circ\overline{C}$ rewrites in the corresponding rewrite theory to $\overline{C'}$ (which is a normal form). Thus, to enable the resulting rewrite system on a given configuration, one needs to first mark the configuration to be reduced (by placing a $\circ$ in front of it) and then to let it rewrite to its normal form. Since the one-step reduction always terminates, the corresponding rewrite task also terminates.

If the original small-step SOS had structural identities, then we translate them into equations in a straightforward manner: each identity $t \equiv t'$ is translated into an equation $\overline{t} = \overline{t'}$. The only difference between the original structural identity and the resulting equation is that the meta-variables of

sorts:
ExtendedConfiguration
subsorts:
Configuration $<$ *ExtendedConfiguration*
operations:
$\circ\_$: *Configuration* $\rightarrow$ *ExtendedConfiguration* // reduce one step
$\star\_$: *Configuration* $\rightarrow$ *ExtendedConfiguration* // reduce all steps
rule:
$\star\, Cfg \rightarrow \star\, Cfg'$ **if** $\circ\, Cfg \rightarrow Cfg'$ // where Cfg, Cfg' are variables of sort *Configuration*

Figure 3.17: Representing small-step and many-step SOS reductions in rewriting logic.

the former become variables in the latter. The role of the two is the same in their corresponding frameworks and whatever we can do with one in one framework we can equivalently do with the other in the other framework; consequently, to simplify the notation and the presentation, we will make abstraction of structural identities and equations in our theoretical developments in the reminder of this chapter.

To obtain the reflexive and transitive many-step closure of the small-step SOS relation in the resulting rewrite setting and thus to be able to obtain an interpreter for the defined language when executing the rewrite system, we need to devise some mechanism to iteratively apply the one-step reduction step captured by rewriting as explained above. There could be many ways to do that, but one simple and uniform way is to add a new configuration marker, say $\star\overline{C}$, with the meaning that $\overline{C}$ must be iteratively reduced, small-step after small-step, either forever or until an irreducible configuration is reached. Figure 3.17 shows how one can define both configuration markers algebraically (assuming some existing *Configuration* sort, e.g., the one in Figure 3.13). To distinguish the marked configurations from the usual configurations and to also possibly allow several one-step markers at the same time, e.g., $\circ\circ\circ\,\overline{C}$, which could be useful for debugging/tracing reasons, we preferred to define the sort of marked configurations as a supersort of *Configuration*. Note that the rule in Figure 3.17 indeed gives $\star$ its desired reflexive and transitive closure property (the reflexivity follows from the fact that the rewrite relation in rewriting logic is reflexive, so $\star C \rightarrow \star C$ for any configuration term C).

Theorem 4. ***(Faithful embedding of small-step SOS into rewriting logic)*** *For any small-step SOS* SmallStep*, and any* SmallStep *appropriate configurations C and C', the following equivalences hold:*

$$\text{SmallStep} \vdash C \rightarrow C' \iff \mathcal{R}_{\text{SmallStep}} \vdash \circ\,\overline{C} \rightarrow^1 \overline{C'} \iff \mathcal{R}_{\text{SmallStep}} \vdash \circ\,\overline{C} \rightarrow \overline{C'}$$

$$\text{SmallStep} \vdash C \rightarrow^\star C' \iff \mathcal{R}_{\text{SmallStep}} \vdash \star\,\overline{C} \rightarrow \star\,\overline{C'}$$

where $\mathcal{R}_{\text{SmallStep}}$ is the rewriting logic semantic definition obtained from SmallStep *by translating each rule in* SmallStep *as above. (Recall from Section 2.7 that $\rightarrow^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewriting logic. Also, as C and C' are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{C'}$ are ground terms.)*

Except for transforming parameters into variables, the only apparent difference between SmallStep and $\mathcal{R}_{\text{SmallStep}}$ is that the latter marks (using $\circ$) all the left-hand side configura-

tions and, naturally, uses conditional rewrite rules instead of conditional deduction rules. As Theorem 4 shows, there is a step-for-step correspondence between their corresponding computations (or executions, or derivations). Therefore, similarly to the big-step SOS representation in rewriting logic, the rewrite theory $\mathcal{R}_{\text{SmallStep}}$ *is* the small-step SOS SmallStep, and *not* an encoding of it.

Recall from Section 3.2.3 that in the case of big-step SOS there were some subtle differences between the one-step $\rightarrow^1$ (obtained by dropping the reflexivity and transitivity rules of rewriting logic) and the usual $\rightarrow$ relations in the rewrite theory corresponding to the big-step SOS. The approach followed in this section based on marking configurations, thus keeping the left-hand and the right-hand sides always distinct, eliminates all the differences between the two rewrite relations in the case of the one-step reduction (the two relations are identical on the terms of interest). The second equivalence in Theorem 4 tells us that we can turn the rewriting logic representation of the small-step SOS language definition into an interpreter by simply marking the configuration to be completely reduced with a $\star$ and then letting the rewrite engine do its job.

It is worthwhile noting that like in the case of the big-step SOS representation in rewriting logic, unfortunately, $\mathcal{R}_{\text{SmallStep}}$ lacks the main strengths of rewriting logic: in rewriting logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{SmallStep}}$ can only apply at the top, sequentially. This should not surprise because, as stated, $\mathcal{R}_{\text{SmallStep}}$ *is* SmallStep, with all its strengths and limitations. By all means, both the $\mathcal{R}_{\text{SmallStep}}$ above and the $\mathcal{R}_{\text{BigStep}}$ in Section 3.2.3 are rather poor-style rewriting logic specifications. However, that is normal, because neither big-step SOS nor small-step SOS were meant to have the capabilities of rewriting logic w.r.t. context-insensitivity and parallelism; since their representations in rewriting logic are faithful, one should not expect that they inherit the additional capabilities of rewriting logic (if they did, then the representations would not be step-for-step faithful, so something would be wrong).

Small-Step SOS of IMP in Rewriting Logic

Figure 3.18 gives the rewriting logic theory $\mathcal{R}_{\text{SmallStep(IMP)}}$ that is obtained by applying the procedure above to the small-step SOS of IMP, namely the formal system SmallStep(IMP) presented in Figures 3.14 and 3.15. As usual, we used the rewriting logic convention that variables start with upper-case letters, and like in the rewrite theory corresponding to the big-step SOS of IMP in Figure 3.8, we used σ (a larger σ symbol) for variables of sort *State*. Besides the parameter vs. variable subtle (but not unexpected) aspect, the only perceivable difference between SmallStep(IMP) and $\mathcal{R}_{\text{SmallStep(IMP)}}$ is the different notational conventions they use. The following corollary of Theorem 4 establishes the faithfulness of the representation of the small-step SOS of IMP in rewriting logic:

Corollary 3. $\text{SmallStep(IMP)} \vdash C \rightarrow C' \iff \mathcal{R}_{\text{SmallStep(IMP)}} \vdash \circ\overline{C} \rightarrow \overline{C'}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system SmallStep(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{SmallStep(IMP)}}$; the two are faithfully equivalent.

☆ Maude Definition of IMP Small-Step SOS

Figure 3.19 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{SmallStep(IMP)}}$ in Figure 3.18, including representations of the algebraic signatures of small-step SOS configurations in Figure 3.13 and of their extensions in Figure 3.17, which are needed to capture small-step SOS

$$\circ\langle X,\sigma\rangle \to \langle \sigma(X),\sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot$$

$$\circ\langle A_1 + A_2,\sigma\rangle \to \langle A_1' + A_2,\sigma\rangle \ \textbf{if}\ \circ\langle A_1,\sigma\rangle \to \langle A_1',\sigma\rangle$$
$$\circ\langle A_1 + A_2,\sigma\rangle \to \langle A_1 + A_2',\sigma\rangle \ \textbf{if}\ \circ\langle A_2,\sigma\rangle \to \langle A_2',\sigma\rangle$$
$$\circ\langle I_1 + I_2,\sigma\rangle \to \langle I_1\ +_{Int}\ I_2,\sigma\rangle$$

$$\circ\langle A_1 \,/\, A_2,\sigma\rangle \to \langle A_1' \,/\, A_2,\sigma\rangle \ \textbf{if}\ \circ\langle A_1,\sigma\rangle \to \langle A_1',\sigma\rangle$$
$$\circ\langle A_1 \,/\, A_2,\sigma\rangle \to \langle A_1 \,/\, A_2',\sigma\rangle \ \textbf{if}\ \circ\langle A_2,\sigma\rangle \to \langle A_2',\sigma\rangle$$
$$\circ\langle I_1 \,/\, I_2,\sigma\rangle \to \langle I_1\ /_{Int}\ I_2,\sigma\rangle \ \textbf{if}\ I_2 \neq 0$$

$$\circ\langle A_1 \texttt{<=} A_2,\sigma\rangle \to \langle A_1' \texttt{<=} A_2,\sigma\rangle \ \textbf{if}\ \circ\langle A_1,\sigma\rangle \to \langle A_1',\sigma\rangle$$
$$\circ\langle I_1 \texttt{<=} A_2,\sigma\rangle \to \langle I_1 \texttt{<=} A_2',\sigma\rangle \ \textbf{if}\ \circ\langle A_2,\sigma\rangle \to \langle A_2',\sigma\rangle$$
$$\circ\langle I_1 \texttt{<=} I_2,\sigma\rangle \to \langle I_1\ \leq_{Int}\ I_2,\sigma\rangle$$

$$\circ\langle\, \texttt{not}\ B,\sigma\rangle \to \langle\, \texttt{not}\ B',\sigma\rangle \ \textbf{if}\ \circ\langle B,\sigma\rangle \to \langle B',\sigma\rangle$$
$$\circ\langle\, \texttt{not true}\,,\sigma\rangle \to \langle\, \texttt{false}\,,\sigma\rangle$$
$$\circ\langle\, \texttt{not false}\,,\sigma\rangle \to \langle\, \texttt{true}\,,\sigma\rangle$$

$$\circ\langle B_1 \,\texttt{and}\, B_2,\sigma\rangle \to \langle B_1' \,\texttt{and}\, B_2,\sigma\rangle \ \textbf{if}\ \circ\langle B_1,\sigma\rangle \to \langle B_1',\sigma\rangle$$
$$\circ\langle\, \texttt{false and}\, B_2,\sigma\rangle \to \langle\, \texttt{false}\,,\sigma\rangle$$
$$\circ\langle\, \texttt{true and}\, B_2,\sigma\rangle \to \langle B_2,\sigma\rangle$$

$$\circ\langle X \texttt{ := } A,\sigma\rangle \to \langle X \texttt{ := } A',\sigma\rangle \ \textbf{if}\ \circ\langle A,\sigma\rangle \to \langle A',\sigma\rangle$$
$$\circ\langle X \texttt{ := } I,\sigma\rangle \to \langle\, \texttt{skip}\,,\sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot$$

$$\circ\langle S_1 \texttt{ ; } S_2,\sigma\rangle \to \langle S_1' \texttt{ ; } S_2,\sigma'\rangle \ \textbf{if}\ \circ\langle S_1,\sigma\rangle \to \langle S_1',\sigma'\rangle$$
$$\circ\langle\, \texttt{skip ; } S_2,\sigma\rangle \to \langle S_2,\sigma\rangle$$

$$\circ\langle\, \texttt{if}\, B\, \texttt{then}\, S_1\, \texttt{else}\, S_2,\sigma\rangle \to \langle\, \texttt{if}\, B'\, \texttt{then}\, S_1\, \texttt{else}\, S_2,\sigma\rangle \ \textbf{if}\ \circ\langle B,\sigma\rangle \to \langle B',\sigma\rangle$$
$$\circ\langle\, \texttt{if true then}\, S_1\, \texttt{else}\, S_2,\sigma\rangle \to \langle S_1,\sigma\rangle$$
$$\circ\langle\, \texttt{if false then}\, S_1\, \texttt{else}\, S_2,\sigma\rangle \to \langle S_2,\sigma\rangle$$

$$\circ\langle\, \texttt{while}\, B\, \texttt{do}\, S,\sigma\rangle \to \langle\, \texttt{if}\, B\, \texttt{then}\, (S \texttt{ ; while}\, B\, \texttt{do}\, S)\, \texttt{else skip}\,,\sigma\rangle$$

$$\circ\langle\, \texttt{var}\, Xl \texttt{ ; } S\rangle \to \langle S, (Xl \mapsto 0)\rangle$$

Figure 3.18: $\mathcal{R}_{\text{SmallStep(IMP)}}$: the small-step SOS of IMP in rewriting logic.

```
mod IMP-CONFIGURATIONS-SMALLSTEP is including IMP-SYNTAX + STATE .
  sorts Configuration ExtendedConfiguration .
  subsort Configuration < ExtendedConfiguration .
  op <_,_> : AExp State -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op <_> : Pgm -> Configuration .
  op o_ : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  op *_ : Configuration -> ExtendedConfiguration [prec 80] .   --- all steps
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-SMALLSTEP is including IMP-CONFIGURATIONS-SMALLSTEP .
  var X : Id .  var Sigma Sigma' : State .  var I I1 I2 : Int .  var Xl : List{Id} .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 : BExp .  var S S1 S1' S2 : Stmt .

 crl o < X,Sigma > => < Sigma(X),Sigma > if Sigma(X) =/=Bool undefined .

 crl o < A1 + A2,Sigma > => < A1' + A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < A1 + A2,Sigma > => < A1 + A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
  rl o < I1 + I2,Sigma > => < I1 +Int I2,Sigma > .

 crl o < A1 / A2,Sigma > => < A1' / A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < A1 / A2,Sigma > => < A1 / A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
 crl o < I1 / I2,Sigma > => < I1 /Int I2,Sigma > if I2 =/=Bool 0 .

 crl o < A1 <= A2,Sigma > => < A1' <= A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < I1 <= A2,Sigma > => < I1 <= A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
  rl o < I1 <= I2,Sigma > => < I1 <=Int I2,Sigma > .

 crl o < not B,Sigma > => < not B',Sigma > if o < B,Sigma > => < B',Sigma > .
  rl o < not  true,Sigma > => < false,Sigma > .
  rl o < not false,Sigma > => <  true,Sigma > .

 crl o < B1 and B2,Sigma > => < B1' and B2,Sigma > if o < B1,Sigma > => < B1',Sigma > .
  rl o < false and B2,Sigma > => < false,Sigma > .
  rl o <  true and B2,Sigma > => < B2,Sigma > .

 crl o < X := A,Sigma > => < X := A',Sigma > if o < A,Sigma > => < A',Sigma > .
 crl o < X := I,Sigma > => < skip,Sigma[I / X] > if Sigma(X) =/=Bool undefined .

 crl o < S1 ; S2,Sigma > => < S1' ; S2,Sigma' > if o < S1,Sigma > => < S1',Sigma' > .
  rl o < skip ; S2,Sigma > => < S2,Sigma > .

 crl o < if B then S1 else S2,Sigma > => < if B' then S1 else S2,Sigma >
  if o < B,Sigma > => < B',Sigma  > .
  rl o < if  true then S1 else S2,Sigma > => < S1,Sigma > .
  rl o < if false then S1 else S2,Sigma > => < S2,Sigma > .

  rl o < while B do S,Sigma > => < if B then (S ; while B do S) else skip,Sigma > .

  rl o < var Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.19: The small-step SOS of IMP in Maude, including the definition of configurations.

in rewriting logic. The Maude module `IMP-SEMANTICS-SMALLSTEP` in Figure 3.19 is executable, so Maude, through its rewriting capabilities, yields a small-step SOS interpreter for IMP the same way it yielded a big-step SOS interpreter in Section 3.2.3; for example, the command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 7132 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip,n |-> 0 , s |-> 5050 >
```

Like for the big-step SOS definition in Maude, one can also use any of the general-purpose tools provided by Maude on the small-step SOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. However, a relatively large number of states will be explored, 1509, due to the non-deterministic evaluation strategy of the various language constructs:

```
Solution 1 (state 1508)
states: 1509  rewrites: 8732 in ... cpu (... real) (0 rewrites/second)
Cfg:ExtendedConfiguration --> * < skip,n |-> 0 & s |-> 5050 >
```

3.3.4 Notes

Small-step structural operational semantics was introduced as just *structural operational semantics* (SOS; no "small-step" qualifier at that time) by Plotkin in a 1981 technical report (University of Aarhus Technical Report DAIMI FN-19, 1981) that included his lecture notes of a programming language course [70]. For more than 20 years this technical report was cited as the main SOS reference by hundreds of scientists who were looking for mathematical rigor in their programming language research. It was only in 2004 that Plotkin's SOS was finally published in a journal [71].

Small-step SOS is pedagogically discussed in several textbooks, two early notable ones being Hennessy [36] (1990) and Winskel [98] (1993). Hennessy [36] uses the same notation as Plotkin, but Winskel [98] prefers a different one to make it clear that it is a one step semantics: $\langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle$. Like for big-step SOS, there is no well-established notation for small-step SOS sequents. There is a plethora of research projects and papers that explicitly or implicitly take SOS as *the* formal language semantics framework. Also, SOS served as a source of inspiration, or of problems to be fixed, to other semantic framework designers, including the author. There is simply too much work on SOS, using it, or modifying it, to attempt to cover it here. We limit ourselves to directly related research focused on capturing SOS as a methodological fragment of rewriting logic.

The marked configuration style that we adopted in this section to faithfully represent small-step SOS in rewriting logic was borrowed from Șerbănuță *et al.* [85]; there, the configuration marker "$\circ$" was called a "configuration modifier". An alternative way to keep the left-hand and the right-hand side configurations distinct was proposed by Meseguer and Braga in [50, 16] in the context of representing MSOS into rewrite logic (see Section 3.6); the idea there was to use two different types of configuration wrappers, one for the left-hand side of the transitions and one for the right-hand side, yielding rewriting logic rules of the form:

$$\{C\} \rightarrow [C'] \;\; \text{if} \;\; \{C_1\} \rightarrow [C'_1] \;\wedge\; \{C_2\} \rightarrow [C'_2] \;\wedge \cdots \wedge \{C_n\} \rightarrow [C'_n].$$

The solution proposed by Meseguer and Braga in [50, 16] builds upon experience with a previous representation of MSOS in rewriting logic in [17] as well as with an implementation of it in Maude [15, 18], where the necessity of being able to inhibit the default reflexivity and transitivity of the rewrite relation took shape. We preferred to follow the configuration modifier approach proposed by Șerbănuță *et al.* [85] because it appears to be slightly less intrusive (we only tag the already existing left-hand terms of rules) and more general (the left-hands of rules can have any structure, not only configurations, including no structure at all, as it happens in most of the rules of reduction semantics with evaluation contexts—see Section 3.7, e.g., Figure 3.40).

Vardejo and Martì-Oliet [95] give a Maude implementation of a small-step SOS definition for a simple imperative language similar to our IMP (Hennessy's *WhileL* language [36]), in which they do not attempt to prevent the inherent transitivity of rewriting. While they indeed obtain an executable semantics that is reminiscent of the original small-step SOS of the language, they actually define directly the transitive closure of the small-step SOS relation; they explicitly disable the reflexive closure by checking $C \neq C'$ next to rewrites $C \to C'$ in rule conditions. A small-step SOS of a simple functional language (Hennessy's *Fpl* language [36]) is also given in [95], following a slightly different style, which avoids the problem above. They successfully inhibit rewriting's inherent transitivity in their definition by using a rather creative rewriting representation style for sequents. More precisely, they work with sequents which appear to the user as having the form $\sigma \vdash a \to a'$, where σ is a state and a, a' are arithmetic expressions, etc., but they actually are rewrite relations between terms $\sigma \vdash a$ and a' (an appropriate signature to allow that to parse is defined). Indeed, there is no problem with the automatic reflexive/transitive closure of rewriting here because the LHS and the RHS of each rewrite rule have different structures. The simple functional language in [95] was pure (no side effects), so there was no need to include a resulting state in the RHS of their rules; if the language had side effects, then this Vardejo and Martì-Oliet's representation of small-step SOS sequents in [95] would effectively be the same as the one by Meseguer and Braga in [50, 16] (possibly using different notations, which is irrelevant).

3.3.5 Exercises

Prove the following exercises, all referring to the IMP small-step SOS in Figures 3.14 and 3.15.

Exercise 44. *Change the small-step rules for* `/` *so that it short-circuits when* a_1 *evaluates to* 0.

Exercise 45. *Change the small-step SOS of the* IMP *conjunction so that it is not short-circuited.*

Exercise 46. *One can rightfully argue that the arithmetic expression in an assignment should not be reduced any step when the assigned variable is not declared. Change the small-step SOS of* IMP *to only reduce the arithmetic expression when the assigned variable is declared.*

Exercise 47. *A sophisticated language designer could argue that the reduction of the assignment statement to* `skip` *is an artifact of using small-step SOS, therefore an artificial and undesired step which affects the intended computational granularity of the language. Change the small-step SOS of* IMP *to eliminate this additional small-step.*
Hint: Follow the style in Exercise 52; note, however, that that style will require more rules and more types of configurations, so from that point of view is more complex.

Exercise 48. *Give a proof system for deriving "terminated configuration" sequents* $C\surd$.

Exercise 49. *One could argue that our small-step SOS rules for the conditional waste a computational step when switching to one of the two branches once the condition is evaluated.*

1. *Give an alternative small-step SOS for the conditional which does not require a computational step to switch to one of the two branches.*

2. *Can one do better than that? That is, can one save an additional step by reducing the corresponding branch one step at the same time with reducing the condition to true or false in one step? Hint: one may need terminated configurations, like in Exercise 48.*

Exercise 50. *Give an alternative small-step SOS definition of* `while` *which wastes no computational step. Hint: do a case analysis on b, like in the rules for the conditional.*

Exercise 51. *Give an alternative small-step SOS definition of variable declarations which wastes no computational steps. Hint: one may need terminated configurations, like in Exercise 48.*

Exercise 52. *Modify the small-step SOS definition of* IMP *such that the configurations in the right-hand sides of the transition sequents are minimal (they should contain both a fragment of program and a state only when absolutely needed). What are the drawbacks of this minimalistic approach, compared to the small-step SOS semantics that we chose to follow?*

Exercise 53. *Show that the small-step SOS resulting from Exercise 52 is equivalent to the one in Figure 3.14 on arithmetic and Boolean expressions, that is, $\langle a, \sigma\rangle \to \langle a', \sigma\rangle$ is derivable with the proof system in Figure 3.14 if and only if $\langle a, \sigma\rangle \to \langle a'\rangle$ is derivable with the proof system in Exercise 52, and similarly for Boolean expressions. However, show that the equivalence does not hold true for statements.*

Exercise 54. *Add* `error` *values and statements, and modify the small-step semantics in Figures 3.14 and 3.15 to allow derivations of sequents whose right-hand side configurations contain* `error` *as their syntactic component. Also, experiment with more meaningful error messages.*

Exercise 55. *For any* IMP *statements s_1, s_1', s_2, s_3 and any states σ, σ', the following hold:*

1. $\textsc{SmallStep}(\text{IMP}) \vdash \langle (\,\texttt{skip}\,;\, s_2)\,;\, s_3, \sigma\rangle \to \langle s_2\,;\, s_3, \sigma\rangle$ *and*
   $\textsc{SmallStep}(\text{IMP}) \vdash \langle\, \texttt{skip}\,;\, (s_2\,;\, s_3), \sigma\rangle \to \langle s_2\,;\, s_3, \sigma\rangle$*; and*

2. $\textsc{SmallStep}(\text{IMP}) \vdash \langle (s_1\,;\, s_2)\,;\, s_3, \sigma\rangle \to \langle (s_1'\,;\, s_2)\,;\, s_3, \sigma'\rangle$ *if and only if*
   $\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1\,;\, (s_2\,;\, s_3), \sigma\rangle \to \langle s_1'\,;\, (s_2\,;\, s_3), \sigma'\rangle$.

Consequently, the following also hold (prove them by structural induction on s_1):
$\textsc{SmallStep}(\text{IMP}) \vdash \langle (s_1\,;\, s_2)\,;\, s_3, \sigma\rangle \to^\star \langle s_2\,;\, s_3, \sigma'\rangle$ *if and only if*
$\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1\,;\, (s_2\,;\, s_3), \sigma\rangle \to^\star \langle s_2\,;\, s_3, \sigma'\rangle$ *if and only if*
$\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1, \sigma\rangle \to^\star \langle\, \texttt{skip}, \sigma'\rangle$.

Exercise 56. *With the* $\textsc{SmallStep}(\text{IMP})$ *proof system in Figures 3.14, 3.15, and 3.16, configuration C terminates iff* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R$ *for some irreducible configuration R.*

Exercise 57. *The small-step SOS of* IMP *is globally deterministic: if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R$ *and* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R'$ *for irreducible configurations R and R', then $R = R'$. Show the same result for the proof system detecting division-by-zero as in Exercise 54.*

Exercise 58. *Show that if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star \langle i, \sigma \rangle$ *for some configuration* C*, integer* i*, and state* σ*, then* C *must be of the form* $\langle a, \sigma \rangle$ *for some arithmetic expression* a*. Show a similar result for Boolean expressions. For statements, show that if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star \langle \texttt{skip}, \sigma \rangle$ *then* C *must be either of the form* $\langle s, \sigma' \rangle$ *for some statement* s *and some state* σ'*, or of the form* $\langle p \rangle$ *for some program* p*.*

Exercise 59. *Prove Theorem 3.*

Exercise 60. *State and show a result similar to Theorem 3 but for the small-step and big-step SOS proof systems in Exercises 54 and 41, respectively.*

3.4 Denotational Semantics

Denotational semantics, also known as *fixed-point semantics*, associates to each syntactically well-defined fragment of program a well-defined, rigorous *mathematical object*. This mathematical object denotes the complete behavior of the fragment of program, no matter in what context it will be used. In other words, the denotation of a fragment of program represents its contribution to the meaning of any program containing it. In particular, equivalence of programs or fragments is immediately translated into equivalence of mathematical objects. The later can be then shown using the entire arsenal of mathematics, which is supposedly better understood and more well-established than that of the relatively much newer field of programming languages. There are no theoretical requirements on the nature of the mathematical domains in which the fragments of program are interpreted, though a particular approach became quite well-established, to an extent that it is by many identified with denotational semantics itself: choose the domains to be appropriate bottomed complete partial orders (abbreviated BCPOs; see Section 2.9), and give the denotation of recursive language constructs (including loops, recursive functions, recursive data-structures, recursive types, etc.) as least fixed-points, which exist thanks to Theorem 1.

Each language requires customized denotational semantics, the same way each language required customized big-step or small-step structural operational semantics in the previous sections in this chapter. For the sake of concreteness, below we discuss general denotational semantics notions and notations by means of our running example language, IMP, without attempting to completely define it. Note that the complete denotational semantics of IMP is listed in Figure 3.20 in Section 3.4.1. Consider, for example, arithmetic expressions in IMP (which are side-effect free). Each arithmetic expression can be thought of as the mathematical object which is a partial function taking a state to an integer value, namely the value that the expression evaluates to in the given state. It is a partial function and not a total one because the evaluation of some arithmetic expressions may not be defined in some states, for example due to illegal operations such as division by zero. Thus, we can define the *denotation* of arithmetic expressions as a *total* function

$$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$$

taking arithmetic expressions to *partial* functions from states to integer numbers. As in all the semantics discussed in this chapter, states are themselves partial maps from names to values. In what follows we will follow a common notational simplification and will write $[\![a]\!]\sigma$ instead of $[\![a]\!](\sigma)$ whenever $a \in AExp$ and $\sigma \in State$, and similarly for other syntactic or semantic categories. To avoid ambiguity in the presence of multiple denotation functions, many works on denotational semantics tag the denotation functions with their corresponding syntactic categories, e.g., $AExp[\![\_]\!]$ or $[\![\_]\!]_{AExp}$. Our IMP language is simple enough that we prefer not to add such tags.

Denotation functions are defined inductively, over the structure of the language constructs. For example, if $i \in Int$ then $[\![i]\!]$ is the constant function i, that is, $[\![i]\!]\sigma = i$ for any $\sigma \in State$. Similarly, if $x \in Id$ then $[\![x]\!]\sigma = \sigma(x)$. As it is the case in mathematics, if an undefined value is used to calculate another value, then the resulting value is also undefined. In particular, $[\![x]\!]\sigma$ is undefined when $x \notin Dom(\sigma)$. The denotation of compound constructs is defined in terms of the denotations of the parts. In other words, we say that denotational semantics is *compositional*. For example, $[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma +_{Int} [\![a_2]\!]\sigma$ for any $a_1, a_2 \in AExp$ and any $\sigma \in State$. For the same reason as above, if any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined then $[\![a_1 + a_2]\!]\sigma$ is also implicitly undefined. One can also chose to explicitly keep certain functions undefined in certain states, such as the denotation of division in

those states in which the denominator is zero:

$$[\![a_1 \,/\, a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \;/_{Int}\; [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

Note that even though the case where $[\![a_2]\!]\sigma$ is undefined ($\bot$) was not needed to be explicitly listed above (because it falls under the first case, $[\![a_2]\!]\sigma \neq 0$), it is still the case that $[\![a_1 \,/\, a_2]\!]\sigma$ is undefined whenever any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined.

An immediate use of denotational semantics is to prove properties about programs. For example, we can show that the addition operation on *AExp* whose denotational semantics was given above is associative. Indeed, we can prove for any $a_1, a_2, a_3 \in AExp$ the following equality of partial functions

$$[\![(a_1 + a_2) + a_3]\!] = [\![a_1 + (a_2 + a_3)]\!]$$

using conventional mathematical reasoning and the fact that the sum $+_{Int}$ in the *Int* domain is associative (see Exercise 61). Note that denotational semantics allows us not only to prove properties about programs or fragments of programs relying on properties of their mathematical domains of interpretation, but also, perhaps even more importantly, it allows us to elegantly formulate such properties. Indeed, what does it mean for a language construct to be associative or, in general, for any desired property over programs or fragments of programs to hold? While one could use any of the operational semantics discussed in this chapter to answer this question, denotational semantics gives us one of the most direct means to state and prove program properties.

Each syntactic category is interpreted into its corresponding mathematical domain. For example, the denotations of Boolean expressions and of statements are total functions of the form:

$$[\![\_]\!] : \; BExp \to (State \rightharpoonup Bool)$$
$$[\![\_]\!] : \; Stmt \to (State \rightharpoonup State)$$

The former is similar to the one for arithmetic expressions above, so we do not discuss it here. The latter is more interesting and deserves to be detailed. Statements can indeed be regarded as partial functions taking states into resulting states. In addition to partiality due to illegal operations in expressions that statements may involve, such as division by zero, partiality in the denotation of statements may also occur for another important reason: *loops may not terminate*. For example, the statement `while (`x`<=`y`) do skip` will not terminate in those states in which the value that x denotes is less than or equal to that of y. Mathematically, we say that the function from states to states that this loop statement denotes is undefined in those states in which the loop statement does not terminate. This will be elaborated shortly, after we discuss other statement constructs.

Since `skip` does not change the state, its denotation is the identity function, i.e., $[\![\texttt{skip}]\!] = 1_{State}$. The assignment statement updates the given state when defined in the assigned variable, that is, $[\![x := a]\!]\sigma = \sigma[[\![a]\!]\sigma/x]$ when $\sigma(x) \neq \bot$ and $[\![a]\!]\sigma \neq \bot$, and $[\![x := a]\!]\sigma = \bot$ otherwise. Sequential composition accumulates the state changes of the denotations of the composed statements, so it is precisely the mathematical composition of the corresponding partial functions: $[\![s_1 \texttt{ ; } s_2]\!] = [\![s_2]\!] \circ [\![s_1]\!]$.

As an example, let us calculate the denotation of the statement "`x := 1; y := 2; x := 3`" when `x` $\neq$ `y`, i.e., the function $[\![\texttt{x := 1; y := 2; x := 3}]\!]$. Applying the denotation of sequential composition twice, we obtain $[\![\texttt{ x := 3 }]\!] \circ [\![\texttt{ y := 2 }]\!] \circ [\![\texttt{ x := 1 }]\!]$. Applying this composed function on a state σ, one gets $([\![\texttt{ x := 3 }]\!] \circ [\![\texttt{ y := 2 }]\!] \circ [\![\texttt{ x := 1 }]\!])\sigma$ equals $\sigma[1/x][2/y][3/x]$ when $\sigma(x)$ and $\sigma(y)$ are both defined, and equals $\bot$ when any of $\sigma(x)$ or $\sigma(y)$ is undefined; let σ' denote

$\sigma[1/x][2/y][3/x]$. By the definition of function update, one can easily see that σ' can be defined as

$$\sigma'(z) = \begin{cases} 3 & \text{if } z = x \\ 2 & \text{if } z = y \\ \sigma(z) & \text{otherwise,} \end{cases}$$

which is nothing but $\sigma[2/y][3/x]$. We can therefore conclude that the statements "`x := 1; y := 2; x := 3`" and "`y := 2; x := 3`" are equivalent, because they have the same denotation.

The denotation of a conditional statement `if` b `then` s_1 `else` s_2 in a state σ is either the denotation of s_1 in σ or that of s_2 in σ, depending upon the denotation of b in σ:

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

The third case above was necessary, because the first two cases do not cover the entire space of possibilities and, in such situations, one may (wrongly in our context here) understand that the function is underspecified in the remaining cases rather than undefined. Using the denotation of the conditional statement above and conventional mathematical reasoning, we can show, for example, that $[\![$ `(if y <= z then x := 1 else x := 2); x := 3` $]\!]$ is the function taking states σ defined in x, y and z to $\sigma[3/x]$.

The language constructs which admit non-trivial and interesting denotational semantics tend to be those which have a recursive nature. One of the simplest such constructs, and the only one we discuss here (see Section 4.8 for other recursive constructs), is IMP's `while` looping construct. Thus, the question we address next is how to define the denotation functions of the form

$$[\![\texttt{while } b \texttt{ do } s]\!] : \mathit{State} \rightharpoonup \mathit{State}$$

where $b \in \mathit{BExp}$ and $s \in \mathit{Stmt}$. What we want is $[\![\texttt{while } b \texttt{ do } s]\!]\sigma = \sigma'$ iff the while loop correctly terminates in state σ' when executed in state σ. Such a σ' may not always exist for two reasons:

1. Because b or s is undefined (e.g., due to illegal operations) in σ or in other states encountered during the loop execution; or

2. Because s (which may contain nested loops) or the while loop itself does not terminate.

If w is the partial function $[\![\texttt{while } b \texttt{ do } s]\!]$, then its most natural definition would appear to be:

$$w(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ w([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

Mathematically speaking, this is a problematic definition for several reasons:

1. The partial function w is defined in terms of itself;

2. It is not clear that such a w exists; and

3. In case it exists, it is not clear that such a w is unique.

To see how easily one can yield inappropriate recursive definitions of functions, we refer the reader to the discussion immediately following Theorem 1, which shows examples of recursive definitions which admit no solutions or which are ambiguous.

We next develop the mathematical machinery needed to rigorously define and reason about partial functions like the w above. More precisely, we frame the mathematics needed here as an instance of the general setting and results discussed in Section 2.9. We strongly encourage the reader to familiarize herself with the definitions and results in Section 2.9 before continuing.

A convenient interpretation of partial functions that may ease the understanding of the subsequent material is as *information* or *knowledge bearers*. More precisely, a partial function $\alpha : \mathit{State} \rightharpoonup \mathit{State}$ can be thought of as carrying knowledge about some states in *State*, namely exactly those on which α is defined. For such a state $\sigma \in \mathit{State}$, the knowledge that α carries is $\alpha(\sigma)$. If α is not defined in a state $\sigma \in \mathit{State}$ then we can think of it as "α *does not have any information about* σ".

Recall from Section 2.9 that the set of partial functions between any two sets can be organized as a bottomed complete partial order (BCPO). In our case, if $\alpha, \beta : \mathit{State} \rightharpoonup \mathit{State}$ then we say that α *is less informative than or as informative as* β, written $\alpha \preceq \beta$, if and only if for any $\sigma \in \mathit{State}$, it is either the case that $\alpha(\sigma)$ is not defined, or both $\alpha(\sigma)$ and $\beta(\sigma)$ are defined and $\alpha(\sigma) = \beta(\sigma)$. If $\alpha \preceq \beta$ then we may also say that β *refines* α or that β *extends* α. Then $(\mathit{State} \rightharpoonup \mathit{State}, \preceq \bot)$ is a BCPO, where $\bot : \mathit{State} \rightharpoonup \mathit{State}$ is the partial function which is undefined everywhere.

One can think of each possible iteration of a while loop as an opportunity to refine the knowledge about its denotation. Before the Boolean expression b of the loop $\texttt{while}\, b\, \texttt{do}\, s$ is evaluated the first time, the knowledge that one has about its denotation function w is the empty partial function $\bot : \mathit{State} \rightharpoonup \mathit{State}$, say w_0. Therefore, w_0 corresponds to no information.

Now suppose that we evaluate the Boolean expression b in some state σ and that it is false. Then the denotation of the while loop should return σ, which suggests that we can refine our knowledge about w from w_0 to the partial function $w_1 : \mathit{State} \rightharpoonup \mathit{State}$, which is an identity on all those states $\sigma \in \mathit{State}$ for which $[\![b]\!]\sigma = \texttt{false}$ and which remains undefined in any other state.

So far we have not considered any state in which the loop needs to evaluate its body. Suppose now that for some state σ, it is the case that $[\![b]\!]\sigma = \texttt{true}$, $[\![s]\!]\sigma = \sigma'$, and $[\![b]\!]\sigma' = \texttt{false}$, that is, that the while loop terminates in one iteration. Then we can extend w_1 to a partial function $w_2 : \mathit{State} \rightharpoonup \mathit{State}$, which, in addition to being an identity on those states on which w_1 is defined, that is $w_1 \preceq w_2$, takes each σ as above to $w_2(\sigma) = \sigma'$.

By iterating this process, one can define a partial function $w_k : \mathit{State} \rightharpoonup \mathit{State}$ for any natural number k, which is defined on all those states on which the while loop terminates in *at most* k evaluations of its Boolean condition (i.e., $k-1$ executions of its body). An immediate property of the partial functions w_0, w_1, w_2, $\ldots$, w_k is that they increasingly refine each other, that is, $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k$. Informally, the partial functions w_k approximate w as k increases; more precisely, for any $\sigma \in \mathit{State}$, if $w(\sigma) = \sigma'$, that is, if the while loop terminates and σ' is the resulting state, then there is some k such that $w_k(\sigma) = \sigma'$. Moreover, $w_n(\sigma) = \sigma'$ for any $n \geq k$.

But the main question still remains unanswered: how to define the denotation $w : \mathit{State} \rightharpoonup \mathit{State}$ of the looping statement $\texttt{while}\, b\, \texttt{do}\, s$? According to the intuitions above, w should be some sort of *limit* of the (infinite) sequence of partial functions $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k \preceq \cdots$. We next formalize all the intuitions above. Let us define the total function

$$\mathcal{F} : (\mathit{State} \rightharpoonup \mathit{State}) \to (\mathit{State} \rightharpoonup \mathit{State})$$

taking partial functions $\alpha : State \rightharpoonup State$ to partial functions $\mathcal{F}(\alpha) : State \rightharpoonup State$ as follows:

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \alpha([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

The partial functions w_k defined informally above can be now rigorously defined as $\mathcal{F}^k(\bot)$, where $\mathcal{F}^k$ stays for k compositions of $\mathcal{F}$, and $\mathcal{F}^0$ is by convention the identity function, i.e., $1_{(State \rightharpoonup State)}$ (which is total). Indeed, one can show by induction on k the following property, where $[\![s]\!]^i$ stays for i compositions of $[\![s]\!] : State \rightharpoonup State$ and $[\![s]\!]^0$ is by convention the identity (total) function on $State$:

$$\mathcal{F}^k(\bot)(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \leq i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \\ & \text{and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \leq j < i \\ \bot & \text{otherwise} \end{cases}$$

We can also show that the following is a chain of partial functions (Exercise 65 requires the reader to prove, for the IMP language, all these facts mentioned above and below)

$$\bot \leq \mathcal{F}(\bot) \leq \mathcal{F}^2(\bot) \leq \cdots \leq \mathcal{F}^n(\bot) \leq \cdots$$

in the BCPO $(State \rightharpoonup State, \leq, \bot)$. As intuitively discussed above, this chain incrementally approximates the desired denotation of $\texttt{while}\, b\, \texttt{do}\, s$. The final step is to realize that $\mathcal{F}$ is a continuous function and thus satisfies the hypotheses of the fixed-point Theorem 1, so we can conclude that the least upper bound (lub) of the chain above, which by Theorem 1 is the least fixed-point $fix(\mathcal{F})$ of $\mathcal{F}$, is the desired denotation of the while loop, that is,

$$[\![\texttt{while}\ b\ \texttt{do}\ s]\!] = fix(\mathcal{F})$$

Remarks. First, note that we indeed want the *least* fixed-point of $\mathcal{F}$, and not some arbitrary fixed-point of $\mathcal{F}$, to be the denotation of the while statement. Indeed, any other fixed-points would define states in which the while loop is intended to be undefined. To be more concrete, consider the simple IMP while loop "`while not(k <= 10) do k := k + 1`" whose denotation is defined only on those states σ with $\sigma(k) \leq 10$ and, on those states, it is the identity. That is,

$$[\![\texttt{while not(k <= 10) do k := k + 1}]\!](\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \leq 10 \\ \bot & \text{otherwise} \end{cases}$$

Consider now another fixed-point $\gamma : State \rightharpoonup State$ of its corresponding $\mathcal{F}$. While γ must still be the identity on those states σ with $\sigma(k) \leq 10$ (indeed, $\gamma(\sigma) = \mathcal{F}(\gamma)(\sigma) = \sigma$ for such $\sigma \in State$), it is not enforced to be undefined on any other states. In fact, it can be shown that the fixed-points of $\mathcal{F}$ are precisely those γ as above with the additional property that $\gamma(\sigma) = \gamma(\sigma')$ for any $\sigma, \sigma' \in State$ with $\sigma(k) > 10$, $\sigma'(k) > 10$, and $\sigma(x) = \sigma'(x)$ for any $x \neq k$. Such a γ can be, for example, the following:

$$\gamma(\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \leq 10 \\ \iota & \text{otherwise} \end{cases}$$

where $\iota \in State$ is some arbitrary but fixed state. It is clear that such γ fixed-points are too informative for our purpose here, since we want the denotation of the while loop to be undefined in

all states in which the loop does not terminate. Any other fixed-point of $\mathcal{F}$ which is strictly more informative that $fix(\mathcal{F})$ is simply too informative.

Second, note that the chain $\bot \leq \mathcal{F}(\bot) \leq \mathcal{F}^2(\bot) \leq \cdots \leq \mathcal{F}^n(\bot) \leq \cdots$ can be stationary in some cases, but in general it is not. For example, when the loop is well-defined and terminates in any state in some fixed maximum number of iterations which does not depend on the state, its denotation is the (total) function in which the chain stabilizes (which in that case is its lub and, by Theorem 1, the fixed-point of $\mathcal{F}$). For example, the chain corresponding to the loop "`while (1 <= k and k <= 10) do k := k + 1`" stabilizes in 12 steps, each step adding more states to the domain of the corresponding partial function until nothing can be added anymore: at step 1 all states σ with $\sigma(k) > 100$ or $\sigma(k) < 1$, at step 2 those with $\sigma(k) = 10$, at step 3 those with $\sigma(k) = 9$, ..., at step 11 those with $\sigma(k) = 1$; then no other state is added at step 12, that is, $\mathcal{F}^{12}(\bot) = \mathcal{F}^{11}(\bot)$. However, the chain associated to a loop is not stationary in general. For example, "`while (k <= 0) do k := k + 1`" terminates in any state, but there is no bound on the number of iterations. Consequently, there is no n such that $\mathcal{F}^n(\bot) = \mathcal{F}^{n+1}(\bot)$. Indeed, the later has strictly more information than the former: $\mathcal{F}^{n+1}$ is defined on all those states σ with $\sigma(k) = -n$, while $\mathcal{F}^n$ is not.

3.4.1 The Denotational Semantics of IMP

Figure 3.20 shows the complete denotational semantics of IMP. There is not much to comment on the denotational semantics of the various IMP language constructs, because they have already been discussed above. Note though that the denotation of conjunction captures the desired short-circuited semantics, in that the second conjunct is evaluated only when the first evaluates to `true`. Also, note that the denotation of programs is still a total function for uniformity (in spite of the fact that some programs may not be well-defined or may not terminate), but one into the BCPO $State_\bot$ (see Section 2.9); thus, the denotation of a program which is not well-defined is $\bot$. Finally, note that, like in the big-step SOS of IMP in Section 3.2.2, we ignore the non-deterministic evaluation strategies of the `+` and `/` arithmetic expression constructs. In fact, since the denotations of the various language constructs are *functions*, non-deterministic constructs cannot be handled in denotational semantics the same way they were handled in operational semantics. There are ways to deal with non-determinism and concurrency in denotational semantics as discussed at the end of this section, but those are more complex and lead to inefficient interpreters when executed, so we do not consider them in this book. We here limit ourselves to denotational semantics of deterministic languages.

3.4.2 Denotational Semantics in Equational/Rewriting Logic

In order to formalize and execute denotational semantics one needs to formalize and execute the fragment of mathematics that is used by the denotation functions. How much mathematics is used is open-ended and is typically driven by the particular programming language in question. Since a denotational semantics associates to each program or fragment of program a mathematical object expressed using the formalized language of the corresponding mathematical domain, the faithfulness of any representation/encoding/implementation of denotational semantics into any formalism directly depends upon the faithfulness of the formalizations of the mathematical domains.

The faithfulness of formalizations of mathematical domains is, however, quite hard to characterize in general. Each mathematical domain formalization may require its own proofs of correctness. Consider, for example, the basic domain of natural numbers. One may choose to formalize it using, e.g., Peano-style equational axioms or λ-calculus (see Section 4.5); nevertheless, none of

Arithmetic expression constructs

$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$

$[\![i]\!]\sigma = i$

$[\![x]\!]\sigma = \sigma(x)$

$[\![a_1 \texttt{ + } a_2]\!]\sigma = [\![a_1]\!]\sigma \ +_{Int} \ [\![a_2]\!]\sigma$

$$[\![a_1 \texttt{ / } a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \ /_{Int} \ [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

Boolean expression constructs

$[\![\_]\!] : BExp \to (State \rightharpoonup Bool)$

$[\![t]\!]\sigma = t$

$[\![a_1 \texttt{ <= } a_2]\!]\sigma = [\![a_1]\!]\sigma \ \leq_{Int} \ [\![a_2]\!]\sigma$

$[\![\texttt{not } b]\!]\sigma = \neg_{Bool}([\![b]\!]\sigma)$

$$[\![b_1 \texttt{ and } b_2]\!]\sigma = \begin{cases} [\![b_2]\!]\sigma & \text{if } [\![b_1]\!]\sigma = \texttt{true} \\ \texttt{false} & \text{if } [\![b_1]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b_1]\!]\sigma = \bot \end{cases}$$

Statement constructs

$[\![\_]\!] : Stmt \to (State \rightharpoonup State)$

$[\![\texttt{skip}]\!]\sigma = \sigma$

$$[\![x \texttt{ := } a]\!]\sigma = \begin{cases} \sigma[[\![a]\!]\sigma \ / \ x] & \text{if } \sigma(x) \neq \bot \\ \bot & \text{if } \sigma(x) = \bot \end{cases}$$

$[\![s_1 \texttt{ ; } s_2]\!]\sigma = [\![s_2]\!][\![s_1]\!]\sigma$

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

$[\![\texttt{while } b \texttt{ do } s]\!] = fix(\mathcal{F}),$ where $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ defined as

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

Programs

$[\![\_]\!] : Pgm \to State_\bot$

$[\![\texttt{var } xl \texttt{ ; } s]\!] = [\![s]\!](xl \mapsto 0)$

Figure 3.20: DENOT(IMP): The denotational semantics of IMP.

these formalizations is powerful enough to mechanically derive any property over natural numbers. We therefore cannot prove faithfulness theorems for our representation of denotational semantics in equational/rewriting logic as we did for the other semantic approaches. Instead, we here limit ourselves to demonstrating our approach using the concrete IMP language and the basic domains of integer numbers and Booleans, together with the mathematical domain already formalized in Section 2.9.5, which allows us to define and execute higher-order functions and fixed-points for them.

Denotational Semantics of IMP in Equational/Rewrite Logic

Figure 3.21 shows a direct representation of the denotational semantics of IMP in Figure 3.20 using the mathematical domain of higher-order functions and fixed-points for them that we formalized in rewriting logic (actually in its membership equational fragment) in Section 2.9.5.

To reduce the number of denotation functions defined, or in other words to simplify our denotational semantics, we chose to collapse all the syntactic categories under only one sort, *Syntax*. Similarly, to reuse existing operations of the CPO domain in Section 2.9.5 (e.g., the substitution) on the new mathematical domains without any additional definitional effort, we collapse all the mathematical domains under CPO; in other words, we now have only one large mathematical domain, CPO, which includes the domains of integer numbers, Booleans and states as subdomains.

There is not much to say about the equations in Figure 3.21; they restate the mathematical definitions in Figure 3.20 using our particular CPO language. The equational formalization of the CPO domain in Section 2.9.5 propagates undefinedness through the CPO operations according to their evaluation strategies. This allows us to ignore some cases, such as the last case in the denotation of the conditional in Figure 3.20; indeed, if $\texttt{app}_{CPO}(\llbracket A \rrbracket, \sigma)$ is undefined in some state σ then $\llbracket X \texttt{ := } A \rrbracket\sigma$ will also be undefined, because $\texttt{app}_{CPO}(..., \bot)$ equals $\bot$ according to our CPO formalization in Section 2.9.5.

☆ Denotational Semantics of IMP in Maude

Figure 3.22 shows the Maude module corresponding to the rewrite theory in Figure 3.21.

3.4.3 Notes

Denotational semantics is the oldest semantic approach to programming languages. The classic paper that introduced denotational semantics as we know it today, then called "mathematical semantics", was published in 1971 by Scott and Strachey [80]. However, Strachey's interest in the subject started much earlier; he published two papers in 1966 and 1967, [88] and [89], respectively, which mark the beginning of denotational semantics. Before Strachey and Scott published their seminal paper [80], Scott published in 1970 a paper which founded what we call today *domain theory* [81]. Initially, Scott formalized domains as complete latices (which also admit a fixed-point theorem); in time, bottomed complete partial orders (BCPOs, see Section 2.9) turned out to have better properties and they eventually replaced the complete latices.

We have only used very simple domains in our semantics of IMP in this section, such as domains of integers, Booleans, and partial functions. Moreover, for simplicity, we are not going to use complex domains for the IMP++ extension in Section 3.5 either. However, complex languages or better semantics may require more complex domains. In fact, choosing the right domains is one of the most important aspects of a denotational semantics. Poor domains may lead to behavioral

sort:

Syntax // generic sort for syntax

subsorts:

AExp, BExp, Stmt, $\mathit{Pgm} < \mathit{Syntax}$ // syntactic categories fall under Syntax

Int, Bool, $\mathit{State} < \mathit{CPO}$ // basic domains are regarded as CPOs

operation:

$[\![\_]\!] : \mathit{Syntax} \to [\mathit{CPO}]$ // denotation of syntax

equations:

// Arithmetic expression constructs:

$[\![I]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ I$

$[\![X]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \sigma(X)$

$[\![A_1\ \texttt{+}\ A_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ (\texttt{app}_{CPO}([\![A_1]\!], \sigma)\ +_{Int}\ \texttt{app}_{CPO}([\![A_2]\!], \sigma))$

$[\![A_1\ \texttt{/}\ A_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \texttt{if}_{CPO}(\ \texttt{app}_{CPO}([\![A_2]\!], \sigma)\ \texttt{=/=}_{Bool}\ 0,$
$\texttt{app}_{CPO}([\![A_1]\!], \sigma)\ /_{Int}\ \texttt{app}_{CPO}([\![A_2]\!], \sigma),\ \ \bot)$

// Boolean expression constructs:

$[\![T]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ T$

$[\![A_1\ \texttt{<=}\ A_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ (\texttt{app}_{CPO}([\![A_1]\!], \sigma)\ \leq_{Int}\ \texttt{app}_{CPO}([\![A_2]\!], \sigma))$

$[\![\texttt{not}\ B]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ (\texttt{not}_{Bool}\ \texttt{app}_{CPO}([\![B]\!], \sigma))$

$[\![B_1\ \texttt{and}\ B_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \texttt{if}_{CPO}(\texttt{app}_{CPO}([\![B_1]\!], \sigma),\ \texttt{app}_{CPO}([\![B_2]\!], \sigma),\ \texttt{false}\ )$

// Statement constructs:

$[\![\texttt{skip}\ ]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \sigma$

$[\![X\ \texttt{:=}\ A]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \texttt{app}_{CPO}(\ \texttt{fun}_{CPO}\ \mathit{arg}\ \texttt{->}\ \texttt{if}_{CPO}(\sigma(X) \neq \bot, \sigma[\mathit{arg}/X], \bot), \texttt{app}_{CPO}([\![A]\!], \sigma))$

$[\![S_1\ \texttt{;}\ S_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \texttt{app}_{CPO}([\![S_2]\!], \texttt{app}_{CPO}([\![S_1]\!], \sigma))$

$[\![\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \texttt{fun}_{CPO}\ \sigma\ \texttt{->}\ \texttt{if}_{CPO}(\texttt{app}_{CPO}([\![B]\!], \sigma), \texttt{app}_{CPO}([\![S_1]\!], \sigma), \texttt{app}_{CPO}([\![S_2]\!], \sigma))$

$[\![\texttt{while}\ B\ \texttt{do}\ S]\!] = \texttt{fix}_{CPO}(\ \texttt{fun}_{CPO}\ \alpha\ \texttt{->}\ \texttt{fun}_{CPO}\ \sigma\ \texttt{->}$
$\texttt{if}_{CPO}(\texttt{app}_{CPO}([\![B]\!], \sigma), \texttt{app}_{CPO}(\alpha, \texttt{app}_{CPO}([\![S]\!], \sigma)), \sigma))$

// Programs:

$[\![\texttt{var}\ Xl\ \texttt{;}\ S]\!] = \texttt{app}_{CPO}([\![S]\!],\ (Xl \mapsto 0))$

Figure 3.21: $\mathcal{R}_{\text{DENOT(IMP)}}$: Denotational semantics of IMP in equational/rewriting logic.

```
mod IMP-SEMANTICS-DENOTATIONAL is including IMP-SYNTAX + STATE + CPO .
  sort Syntax .
  subsorts AExp BExp Stmt Pgm < Syntax .
  subsorts Int Bool State < CPO .

  op [[_]] : Syntax  -> CPO .    --- Syntax interpreted in CPOs

  var X : Id . var Xl : List{Id} .  var I : Int .  var A1 A2 A : AExp .
  var T : Bool .  var B1 B2 B : BExp .  var S1 S2 S : Stmt .
  ops sigma alpha arg : -> CPOVar .

  eq [[I]] = funCPO sigma -> I .

  eq [[X]] = funCPO sigma -> sigma(X) .

  eq [[A1 + A2]] = funCPO sigma -> (appCPO([[A1]],sigma) +Int appCPO([[A2]],sigma)) .

  eq [[A1 / A2]] = funCPO sigma -> ifCPO(appCPO([[A2]],sigma) =/=Bool 0,
                                         appCPO([[A1]],sigma) /Int appCPO([[A2]],sigma),
                                         undefined) .

  eq [[T]] = funCPO sigma -> T .

  eq [[A1 <= A2]] = funCPO sigma -> (appCPO([[A1]],sigma) <=Int appCPO([[A2]],sigma)) .

  eq [[not B]] = funCPO sigma -> (notBool appCPO([[B]],sigma)) .

  eq [[B1 and B2]] = funCPO sigma -> ifCPO(appCPO([[B1]],sigma),appCPO([[B2]],sigma),false) .

  eq [[skip]] = funCPO sigma -> sigma .

  eq [[X := A]]
   = funCPO sigma
     -> appCPO(funCPO arg
               -> ifCPO(sigma(X) =/=Bool undefined, sigma[arg / X], undefined),
               appCPO([[A]],sigma)) .

  eq [[S1 ; S2]] = funCPO sigma -> appCPO([[S2]],appCPO([[S1]],sigma)) .

  eq [[if B then S1 else S2]]
   = funCPO sigma -> ifCPO(appCPO([[B]],sigma),appCPO([[S1]],sigma),appCPO([[S2]],sigma)) .

  eq [[while B do S]]
   = fixCPO(funCPO alpha
            -> funCPO sigma
               -> ifCPO(appCPO([[B]],sigma),appCPO(alpha,appCPO([[S]],sigma)),sigma)) .

  eq [[(var Xl ; S)]] = appCPO([[S]],(Xl |-> 0)) .
endm
```

Figure 3.22: The denotational semantics of IMP in Maude

limitations or to non-modular denotational semantic definitions. There are two additional important contributions to domain theory that are instrumental in making denotational semantics more usable:

- *Continuation domains.* The use of *continuations* in denotational semantics was proposed in 1974, in a paper by Strachey and Wadsworth [90]. Wadsworth was the one who coined the term "continuation", as representing "the meaning of the rest of the program". Continuations allow to have direct access to the execution flow, in particular to modify it, as needed for the semantics of abrupt termination, exceptions, or call/cc (Scheme was the first language to support call/cc). This way, continuations bring modularity and elegance to denotational definitions. However, they come at a price: using continuations affects the entire language definition (so one needs to change almost everything) and the resulting semantics are harder to read and reason about. There are countless uses of continuations in the literature, not only in denotational semantics; we refer the interested reader to a survey paper by Reynolds, which details continuations and their discovery from various perspectives [73].

- *Powerdomains.* The usual domains of partial functions that we used in our denotational semantics of IMP are not sufficient to define non-deterministic and/or concurrent languages. Consider, for example, the denotation of statements, which are partial functions from states to states. If the language is non-deterministic or concurrent, then a statement may take a state into any of many possible different states, or, said differently, it may take a state into a *set* of states. To give denotational semantics to such languages, Plotkin proposed and formalized the notion of *powerdomain* [69]; the elements of a powerdomain are sets of elements of an underlying domain. Powerdomains make it thus possible to give denotational semantics to non-deterministic languages; used in combination with a technique called "resumptions", powerdomains can also be used to give interleaving semantics to concurrent languages.

As seen above, most of the denotational semantics ideas and principles have been proposed and developed in the 1970s. While it is always recommended to read the original papers for historical reasons, some of them may actually use notational conventions and constructions which are not in current use today, making them less accessible. The reader interested in a more modern presentation of denotational semantics and domain theory is referred to Schmidt's denotational semantics book [79], and to Mosses' denotational semantics chapter [59] and Gunter and Scott's domain theory chapter [34] in the Handbook of Theoretical Computer Science (1990).

Denotational semantics are commonly defined and executed using functional languages. A particularly appealing aspect of functional languages is that the domains of partial functions, which are crucial for almost any denotational semantics, and fixed-point operators for them can be very easily defined using the already existing functional infrastructure of these languages; in particular, one needs to define no λ-like calculus as we did in Section 2.9.5. There is a plethora of works on implementing and executing denotational semantics on functional languages. We here only mention Papaspyrou's denotational semantics of C [67] which is implemented in Haskell; it uses about 40 domains in total and spreads over about 5000 lines of Haskell code. An additional advantage of defining denotational semantics in functional languages is that one can relatively easily port them into theorem provers and then prove properties or meta-properties about them. We refer the interested reader to Nipkow [64] for a simple example of how this is done in the context of the Isabelle/HOL prover.

A very insightful exercise is to regard domain theory and denotational semantics through the lenses of *initial algebra semantics* [30], which was proposed by Goguen *et al.* in 1977. The initial

algebra semantics approach is simple and faithful to equational logic (Section 2.3); it can be summarized with the following steps:

1. Define a language syntax as an algebraic signature, say Σ, which admits an initial (term) algebra, say T_Σ;

2. Define any semantic domain of interest as the carrier of corresponding sort in some (domain) algebra, say D;

3. Give D a Σ-algebra structure, by defining operations corresponding to all symbols in Σ;

4. Conclude that the meaning of the language in D is the unique morphism $D_\_ : T_\Sigma \to D$, which gives meaning D_t in D to any fragment of program t.

Let us apply the initial algebra semantics steps above to IMP:

1. Σ is the signature in Figure 3.2;

2. D_{AExp} is the BCPO $(State \rightharpoonup Int, \preceq, \perp)$ and similarly for the other sorts;

3. $D_{\_+\_} : D_{AExp} \times D_{AExp} \to D_{AExp}$ is the (total) function defined as $D_{\_+\_}(f_1, f_2)(\sigma) = f_1(\sigma) +_{Int} f_2(\sigma)$ for all $f_1, f_2 \in D_{AExp}$ and $\sigma \in State$, and similarly for the other syntactic constructs.

4. The meaning of IMP is given by the unique morphism $D_\_ : T_\Sigma \to D_{AExp}$.

Therefore, one can regard a denotational semantics of a language as an initial algebra semantics applied to a *particular* algebra D; in our IMP case, for example, what we defined as $[\![a]\!]$ for $a \in AExp$ in denotational semantics is nothing but D_a. Moreover, we can now *prove* our previous denotational semantics definitions; for example, we can prove $D_{a_1 + a_2}(\sigma) = D_{a_1}(\sigma) +_{Int} D_{a_2}(\sigma)$. The above was a very brief account of initial algebra semantics, but sufficient to appreciate the foundational merits of initial algebra semantics in the context of denotational semantics (initial algebra semantics has many other applications). From an initial algebra semantics perspective, a denotational semantics is all about defining an algebra D in which the syntax is interpreted. How each fragment gets a meaning follows automatically, from more basic principles. In that regard, initial algebra semantics achieves a cleaner separation of concerns: syntax is defined as a signature, and semantics is defined as an algebra. There are no equations mixing syntax and semantics, like $[\![a_1 + a_2]\!] = [\![a_1]\!] +_{Int} [\![a_2]\!]$.

We are not aware of any other approaches to define denotational semantics using rewriting and then executing it on rewrite engines as we did in Section 3.4.2. While this is not difficult in principle, as seen in this section, it requires one to give executable rewriting definitions of semantic domains and of fixed points. This may be a tedious and repetitive process on simplistic rewrite engines; for example, the use of membership equational logic, which allows computations to take place also on terms whose intermediate sorts cannot be determined, was crucial for our formalization in Section 2.9.5. Perhaps the closest approach to ours is the one by Goguen and Malcolm in [33]; they define a simple imperative language using the OBJ system (a precursor of Maude) and a style which is a mixture of initial algebra semantics and operational semantics. For example, no fixed-points are used in [33], the loops being simply unrolled like in small-step SOS (see Section 3.3).

3.4.4 Exercises

Prove the following exercises, all referring to the IMP denotational semantics in Figure 3.20.

Exercise 61. *Show the associativity of the addition expression construct, that is, that*

$$[\![(a_1 + a_2) + a_3]\!] = [\![a_1 + (a_2 + a_3)]\!]$$

for any $a_1, a_2, a_3 \in AExp$.

Exercise 62. *Show the associativity of the statement sequential composition, that is, that*

$$[\![s_1 \texttt{;} (s_2 \texttt{;} s3)]\!] = [\![(s_1 \texttt{;} s_2) \texttt{;} s_3]\!]$$

for any $s_1, s_2, s_3 \in Stmt$. *Compare the elegance of formulating and proving this result using denotational semantics with the similar task using small-step SOS (see Exercise 55).*

Exercise 63. *State and prove the (correct) distributivity property of division over addition.*

Exercise 64. *Prove the equivalence of statements of the form "*$(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ \texttt{;}\ s$*" and "*$\texttt{if}\ b\ \texttt{then}\ (s_1\ \texttt{;}\ s)\ \texttt{else}\ (s_2\ \texttt{;}\ s)$*".*

Exercise* 65. *Prove that the functions* $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ *associated to* IMP *while loops satisfy the hypotheses of the fixed-point Theorem 1, so that the denotation of* IMP *loops is indeed well-defined. Also, prove that the partial functions* $w_k : State \rightharpoonup State$ *defined as*

$$w_k(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \le i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \text{ and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \le j < i \\ \bot & \text{otherwise} \end{cases}$$

are well-defined, that is, that if an i *as above exists then it is unique. Then prove that* $w_k = \mathcal{F}^k(\bot)$.

3.5 IMP++: IMP Extended with Several Features

Our goal here is to challenge the modularity of the basic semantic approaches discussed so far in this chapter, namely big-step SOS, small-step SOS and denotational semantics, by means of a simple programming language design experiment. We shall extend the IMP language in Section 3.1 with several common language features and then attempt to give the resulting language, called IMP++, a formal semantics following each of the approaches. In each case, we aim at reusing the existing semantics of IMP as much as possible. The additional features of IMP++ are the following:

1. A variable increment operation, `++` *Id*, whose role is to infuse side effects in expressions;
2. An input expression construct, `read()` , and an output statement construct, `print(` *AExp* `)` , whose role is to modify the configurations (one needs to add input/output buffers);
3. Abrupt termination, both by means of an explicit `halt` statement and by means of implicit division-by-zero, whose role is to enforce a sudden change of the evaluation context;
4. Spawning a new thread, `spawn` *Stmt*, which executes the given statement concurrently with the rest of the program, sharing all the variables. The role of the `spawn` statement is to test the support of the various semantic approaches for concurrent language features.
5. Blocks allowing local variable declarations, { *Stmt* } where *Stmt* can include declarations of the form `var` **List{** *Id* **}** (regarded as ordinary statements). The scope of local declarations is the reminder of the current block. The introduction of blocks with locals begs for changing some of the existing syntax and semantics. For example, there is no need for the current global variable declarations, because they can be replaced by local declarations. The role of this extension is threefold: (1) to demonstrate how execution environment recovery can be achieved in each semantic approach; (2) to generate some non-trivial feature interactions (e.g., spawned threads share the spawning environment); (3) to highlight a language design scenario where the introduction of a new feature may affect the design of the previous ones.

The criterion used for selecting these new features of IMP++ was twofold: on the one hand, these are quite ordinary features encountered in many languages; on the other hand, each of them exposes limitations of one or more of the conventional semantic approaches in this chapter (both before and after this section). Both IMP and IMP++ are admittedly toy languages. However, if a certain programming language semantical style has difficulties in supporting any of the features of IMP or any of the above IMP extensions in IMP++, then one should most likely expect the same problems, but of course amplified, to occur in practical attempts to define real-life programming languages. By "difficulty" we here also mean lack of modularity, that is, that in order to define a new feature one needs to make unrelated changes in the already existing semantics of other features.

IMP++ extends and modifies IMP both syntactically and semantically. Syntactically, it removes from IMP the global declarations and adds the following constructs:

$$
\begin{array}{rcl}
\mathit{AExp} & ::= & \texttt{++}\,\mathit{Id} \;\mid\; \texttt{read()} \\
\mathit{Stmt} & ::= & \texttt{print(}\,\mathit{AExp}\,\texttt{)} \\
& \mid & \texttt{halt} \\
& \mid & \texttt{spawn}\,\mathit{Stmt} \\
& \mid & \{\} \;\mid\; \{\,\mathit{Stmt}\,\} \;\mid\; \texttt{var}\ \textbf{List\{}\mathit{Id}\textbf{\}} \\
\mathit{Pgm} & ::= & \mathit{Stmt}
\end{array}
$$

Semantically, in addition to defining the new language constructs above, we prefer that division-by-zero implicitly halts the program in IMP++, same like the explicit use of `halt`, but in the middle of an expression evaluation. When such an error takes place, one could also generate an error message; however, for simplicity, we do not consider error messages here, only silent termination.

Before we continue with the details of defining each of the new language features in each of the semantics, we mention that there could be various ways to define these. Our goal in this section is to illustrate the lack of modularity of the various semantic styles in different language extension scenarios, and not necessarily to output good error messages. For example, a program that performs a division by zero simply halts its execution when the division by zero takes place.

We first take the various IMP language extensions one by one, discussing what it takes to add each of them to IMP making abstraction of the other features. To make our language design experiment more realistic, when defining each feature we pretend that we do not know what other features will be added. That is, we attempt to achieve local optima for each feature independently. Then, in Section 3.5.6, we put all the features together in the IMP++ language.

3.5.1 Adding Variable Increment

Like in several main-stream programming languages, `++`x increments the value of x in the state and evaluates to the incremented value. This way, the increment operation makes the evaluation of expressions to now have side effects. Consider, for example, the following two programs:

```
var m, n, s ;
n := 100 ;
while (++ m <= n) do (s := s + m)
```

```
var x ;
x := 1 ;
x := ++ x / (++ x / x)
```

The first program shows that the side effect of variable increment can take place anywhere, even in the condition of the while loop; this is actually quite a common programming pattern in languages with variable increment. The second program shows that the addition of side-effects makes the originally intended evaluation strategies of the various expression constructs important. Indeed, recall that for demonstration purposes we originally wanted `+` and `/` to be non-deterministic (i.e., to evaluate their arguments stepwise non-deterministically, possibly interleaving their evaluations), while `<=` to be left-right sequential. These different evaluation strategies can now lead to different program behaviors. For example, the second program above has no fewer than five different behaviors! Indeed, the expression assigned to x can evaluate to 0, 1, 2, 3, and can also perform a division-by-zero. Unfortunately, not all semantic approaches are able to capture all these behaviors.

Big-Step SOS

Big-step SOS is one of the semantics which is the most affected by the inclusion of side effects in expressions, because the previous triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ need to change to four-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$. These changes are necessary to account for collecting the possible side effects generated by the evaluation of expressions (note that the evaluation of Boolean expressions, because of `<=`, can also have side effects). The big-step SOS of almost all the language constructs needs to change as well. For example, the original big-step SOS of division, namely

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \Downarrow \langle i_1 \ /_{Int} \ i_2\rangle} \quad \text{if } i_2 \neq 0$$

changes as follows:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2\rangle}, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma_2\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1\rangle}, \quad \text{where } i_2 \neq 0$$

The rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. We will shortly explain why they fail to fully capture the desired behaviors. One should similarly consider the side effects of expressions in the semantics of statements that need to evaluate expressions. For example, the semantics of the while loop needs to change to propagate the side effects of its condition both when the loop is taken and when the loop is not taken.

Let us next include the big-step semantics of the increment operation; once all the changes to the existing semantics of IMP are applied, the big-step semantics of increment is straightforward:

$$\langle \texttt{++}\, x, \sigma\rangle \Downarrow \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle \qquad \text{(BigStep-Inc)}$$

Indeed, the problem with big-step is not to define the semantics of variable increment, but what it takes to be able to do it. One needs to redefine configurations as explained above and, consequently, to change the semantics of all the already existing features of IMP to use the new configurations. This, and other features defined later on, show how non-modular big-step semantics is.

In addition to being non-modular, big-step SOS cannot properly deal with non-determinism. While it can capture some limited degree of non-determinism as shown above with `/`, namely it can *non-deterministically choose* which of the subexpressions to evaluate first, it cannot define the full non-deterministic strategy (unless we make radical changes to the definition, such as working with sets of values instead of values, which significantly complicate everything and still fail to capture the non-deterministic behaviors—as it would only capture the non-deterministic evaluation results). To see how the non-deterministic choice evaluation strategy in big-step semantics fails to capture all the desired behaviors, consider the expression $\texttt{++}\, x \mathbin{/} (\texttt{++}\, x \mathbin{/} x)$ with x initially 1, as in the second program at the beginning of Section 3.5.1. This expression can only evaluate to 1, 2 or 3 under non-deterministic choice strategy like we get in big-step SOS. Nevertheless, as explained at the beginning of Section 3.5.1, it could also evaluate to 0 and even perform a division-by-zero under a fully non-deterministic evaluation strategy; we will correctly obtain all these behaviors when using the small-step semantic approaches.

Big-step semantics not only misses behaviors due to its lack of support for non-deterministic evaluation strategies, like shown above, but also hides misbehaviors that it, in principle, detects. For example, assuming $x > 0$, the expression $1 \mathbin{/} (x \mathbin{/} \texttt{++}\, x)$ can either evaluate to 1 or perform an erroneous division by zero. If one searches for all the possible evaluations of a program containing such an expression using the big-step semantics in this section, one will only see the behavior where this expression evaluates to 1; one will never see the erroneous behavior where the division by zero takes place. This will be fixed in Section 3.5.3, where we modify the big-step SOS to support abrupt termination. However, without modifying the semantics, the language designer using big-step semantics may wrongly think that the program is correct. Contrast that with small-step semantics, which, even when one does not add support for abrupt termination, one still detects the wrong behavior by getting stuck on the configuration obtained right before the division by zero.

Additionally, as already explained in Section 3.2.3, the new configurations may be problematic when one wants to execute big-step definitions using rewriting. Indeed, one needs to remove resulting rewrite rules that lead to non-termination, such as rules of the form $R \to R$ corresponding to big-step sequents $R \Downarrow R$ where R are result configurations (e.g., $\langle i, \sigma \rangle$ with $i \in Int$ or $\langle t, \sigma \rangle$ with $t \in \{\texttt{true}, \texttt{false}\}$). We do not use this argument against big-step SOS (its poor modularity is sufficient to disqualify big-step in the competition for an ideal language definitional framework), but rather as a warning to the reader who wants to execute it using rewriting engines (like Maude).

Type System using Big-Step SOS

The typing policy of variable increment is the same as that of variable lookup: provided it has been declared, the incremented variable types to an integer. All we need is to add the following typing rule for increment to the already existing typing rules in Figure 3.10:

$$(xl, x, xl') \vdash \texttt{++}\, x : int \qquad \text{(BigStepTypeSystem-Inc)}$$

Small-Step SOS

Including side effects in expressions is not as bad in small-step semantics as in big-step semantics, because, as discussed in Section 3.3, in small-step SOS one typically uses sequents whose left and right configurations have the same structure even in cases where only some of the configuration components change (e.g., one typically uses sequents of the form $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ instead of $\langle a, \sigma \rangle \to \langle a' \rangle$); thus, expressions, like any other syntactic categories including statements, can seamlessly modify the state if they need to. However, since we deliberately did not anticipate the inclusion of side effects in expression evaluation, we still have to go back through the existing definition and modify *all* the rules involving expressions to propagate the side effects. For example, the small-step rule

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \to \langle a_1' \ / \ a_2, \sigma \rangle}$$

for the first argument of `/` does not apply anymore when the next step in a_1 is an increment operation (since the state σ changes), so it needs to change to

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma' \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \to \langle a_1' \ / \ a_2, \sigma' \rangle}$$

Of course, all these changes due to side-effect propagation would have not been necessary if we anticipated that side effects may be added to the language, but the entire point of this exercise is to study the strengths of the various semantic approaches without knowing what comes next.

Once all the changes are applied, one can define the small-step SOS of the increment operation almost identically to its big-step SOS (increment is one atomic step, so small equals big):

$$\langle \texttt{++}\, x, \sigma \rangle \to \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle \qquad \text{(SmallStep-Inc)}$$

Denotational Semantics

The introduction of expression side effects affects denotational semantics even worse than it affects the big-step SOS. Not only that one has to change the denotation of almost every language construct,

but the changes are also heavier and more error prone than for big-step SOS. The first change that needs to be made is the type of the denotation functions for expressions:

$$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$$
$$[\![\_]\!] : BExp \to (State \rightharpoonup Bool)$$

need to change into

$$[\![\_]\!] : AExp \to (State \rightharpoonup Int \times State)$$
$$[\![\_]\!] : BExp \to (State \rightharpoonup Bool \times State)$$

respectively, to take into account the fact that expressions also return a new possibly changed state besides a result when evaluated. Then one has to change the definitions of the denotation functions for each expression construct to propagate the side effects and to properly extract/combine values and states from/in pairs. For example, the previous denotation function of division, where $[\![a_1 \,/\, a_2]\!]\sigma$ was defined as

$$\begin{cases} [\![a_1]\!]\sigma \;/_{Int}\; [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

needs to change to be defined as

$$\begin{cases} (1^{\text{st}}([\![a_1]\!]\sigma) \;/_{Int}\; 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))), 2^{\text{nd}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) & \text{if } 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) \neq 0 \\ \bot & \text{if } 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) = 0 \end{cases}$$

The above is a bit heavy, repetitive and thus error prone. In implementations of denotational semantics, and sometimes even on paper definitions, one typically uses let binders, or λ-abstractions (see Section 4.5), to bind each subexpression appearing more than once in a denotation function to some variable and then using that variable in each place.

In addition to the denotations of expressions, the denotation functions of all statements except for those of `skip` and sequential composition also need to change, because they involve expressions and need to take their side effects into account. For example, the denotation of the while loop statement `while` b `do` s is the fixed-point of the total function

$$\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$$

defined as

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([\![s]\!](2^{\text{nd}}([\![b]\!]\sigma))) & \text{if } 1^{\text{st}}([\![b]\!]\sigma) = \texttt{true} \\ 2^{\text{nd}}([\![b]\!]\sigma) & \text{if } 1^{\text{st}}([\![b]\!]\sigma) = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

All the changes above were necessary to support the side effects generated by the increment construct. The denotational semantics of programs does not need to change. All programs that make no use of increment should still have exactly the same semantics as before (to be precise, the program denotation functions are different as syntactic terms, but they evaluate to the same values when invoked). We are now ready to give the denotational semantics of increment:

$$[\![\texttt{++}\, x]\!]\sigma = \begin{cases} (\sigma(x) +_{Int} 1, \sigma[\sigma(x) +_{Int} 1/x]) & \text{if } \sigma(x) \neq \bot \\ \bot & \text{if } \sigma(x) = \bot \end{cases} \qquad \text{(DENOTATIONAL-INC)}$$

Like for big-step SOS, giving the denotational semantics of increment is not difficult; the difficulty stays in what it takes to be able to do so.

Needless to say that denotational semantics, as we used it here, is very non-modular. The brute force approach above is the most straightforward approach when one's goal is to exclusively add increment to IMP—recall that our experiment in this section assumes that each language extension is the last one. When one expects many extensions to a language that in an operational setting would yield changes to the program configuration, in denotational semantics one is better served using a *continuation* based or a *monadic* style. These styles were briefly mentioned in Section 3.4.3 and further discussed in Section 3.10. They are more involved, and thus less accessible to non-expert language designers. Moreover, switching to such styles is a radical change to a language definition, which requires a complete redefinition of the language. It is therefore highly recommended that one starts directly with a continuation or monadic style if one expects many and non-trivial language extensions. We here, however, prefer the straightforward denotational approach because it is easier to understand and because our overall focus of this book is more operational than denotational.

Besides lacking modularity, the denotational semantics above also lacks non-determinism. Indeed, note that, for example, our denotation of division first evaluates the first expression and then the second expression. Since expressions have side effects, different orders of evaluation can lead to different behaviors. Since the denotations of expressions are partial functions, they cannot have two different behaviors in the same state; therefore, unlike in big-step SOS, we cannot simply add another equation for the other order of evaluation because that would yield an inconsistent equational theory. The consecrated method to define non-determinism in denotational semantics is to use *powerdomains*, as briefly discussed in Section 3.4.3 and further discussed in Section 3.10. Like using continuations or monads, the use of powerdomains also requires a complete redesign of the entire semantics and makes it less accessible to non-experts. Moreover, one cannot obtain a feasible executable model of the language anymore, because the use of powerdomains requires to collect all possible behaviors of any fragment of program at any point in execution; this will significantly slow down the execution of the semantics, making it, for example, infeasible or even unusable as an interpreter anymore.

3.5.2 Adding Input/Output

The semantics of the input expression construct `read()` is that it consumes the next integer from the "input buffer" and evaluates to that integer. The semantics of the output statement construct `print(`a`)` is that a is first evaluated to some integer, which is then collected in an "output buffer". By a "buffer" we here mean some list structure over integers. The semantics of the input/output constructs will be given in such a way that the input buffer can only be removed integers from its beginning and the output buffer can only be appended integers to its end. If there is no integer left in the input buffer then `read()` blocks. The output buffer is assumed unbounded, so `print(`a`)` never blocks when outputting the value of a. Consider the following two programs:

```
var m, n, s ;
n := read() ;
while (m <= n) do
  ( print(m) ;  s := s + m ;  m := m + 1 ) ;
print(s)
```

```
var s ;
s := 0 ;
while not(read() <= 0) do
  s := s + read() ;
print(s)
```

The first reads one integer i from the beginning of the input buffer and then it appends $i+2$ integers to the end of the output buffer (the numbers 0, 1, 2, ..., i followed by their sum). The second reads a potentially unbounded number of integers from the input buffer, terminating if and only if it reads a non-positive integer on an odd position in the input buffer; when that happens, it outputs the sum of the elements on the even positions in the input buffer up to that point.

It is interesting to note that the addition of `read()` to IMP means that expression evaluation becomes non-deterministic, regardless of whether we have variable increment or not in our language. Indeed, since `/` is non-deterministic, an expression of the form `read() / read()` can evaluate the two reads in any order; for example, if the first two integers in the input buffer are 7 and 3, then this expression can evaluate to either 2 or 0.

Let us formalize buffers. Assume colon-separated integer lists with ϵ as identity, $\mathbf{List}^{\epsilon}_{:}\{Int\}$, and let ω, ω', ω_1, etc., range over such lists of integers. The same way we decided for notational convenience to let *State* be an alias for the map sort $\mathbf{Map}\{Id \mapsto Int\}$ (Section 3.1.2), from here on we also let *Buffer* alias the list sort $\mathbf{List}^{\epsilon}_{:}\{Int\}$.

In a formal language semantics, providing the entire input as part of the initial configuration and collecting the entire output in the result configuration is acceptable, although in implementations of the language one will most likely want the input/output to be interactive. There is some flexibility as to where the input and output buffers should be located in the configuration. One possibility is as new top-level components in the configuration. Another possibility is as special variables in the already existing state. The latter would require some non-trivial changes in the mathematical model of the state, so we prefer to follow the former approach in the sequel. An additional argument in favor of our choice is that sooner or later one needs to add new components to the configuration anyway, so we take this opportunity to discuss how robust/modular the various semantic styles are with regards to changes in the structure of the configuration.

Big-Step SOS

To accommodate the input and the output buffers, all configurations and all sequents we had in the original big-step SOS of IMP in Sections 3.2.1 and 3.2.2 need to change. For example, since expressions can now consume input, the original expression sequents of form $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ need to change into sequents $\langle a, \sigma, \omega\rangle \Downarrow \langle i, \omega'\rangle$ and $\langle b, \sigma, \omega\rangle \Downarrow \langle t, \omega'\rangle$ (recall that we add one feature at a time, so expression evaluation currently does not have side effects on the state), respectively, where $\omega, \omega' \in \mathit{Buffer}$. Also, the big-step SOS rules for expressions need to change to take into account both the new configurations and the fact that expression evaluation can now affect the input buffer. For example, the original big-step SOS of division, namely

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \Downarrow \langle i_1 \ /_{Int} \ i_2\rangle} \quad \text{if } i_2 \neq 0$$

changes as follows:

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle i_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega_1\rangle \Downarrow \langle i_2, \omega_2\rangle}{\langle a_1 \ / \ a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \omega_2\rangle}, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma, \omega_2\rangle \Downarrow \langle i_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega\rangle \Downarrow \langle i_2, \omega_2\rangle}{\langle a_1 \ / \ a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \omega_1\rangle}, \quad \text{where } i_2 \neq 0$$

Like for the variable increment, the rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. Unfortunately, they also only capture a non-deterministic choice strategy, failing to capture the intended full non-determinism of division.

Since statements can both consume input and produce output, their big-step SOS sequents need to change from $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ to $\langle s, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma', \omega'_{in}, \omega_{out}\rangle$, where $\omega_{in}, \omega'_{in} \in \mathit{Buffer}$ are the input

buffers before and, respectively, after the evaluation of statement s, and where $\omega_{out} \in \mathit{Buffer}$ is the output produced during the evaluation of s. Unfortunately, all big-step SOS rules for statements also have to change, to accommodate the additional input and/or output components in configurations. For example, the semantics of sequential composition needs to change from

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle \qquad \langle s_2, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle s_1 \,;\, s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle}$$

to

$$\frac{\langle s_1, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma_1, \omega^1_{in}, \omega^1_{out}\rangle \qquad \langle s_2, \sigma_1, \omega^1_{in}\rangle \Downarrow \langle \sigma_2, \omega^2_{in}, \omega^2_{out}\rangle}{\langle s_1 \,;\, s_2, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma_2, \omega^2_{in}, \omega^1_{out} : \omega^2_{out}\rangle}$$

Note that the outputs of s_1 and of s_2 have been appended in order to yield the output of s_1 ; s_2.

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the big-step SOS rule for programs from

$$\frac{\langle s, xl \mapsto 0\rangle \Downarrow \langle \sigma\rangle}{\langle \texttt{var}\ xl\ ;\ s\rangle \Downarrow \langle \sigma\rangle}$$

into

$$\frac{\langle s, xl \mapsto 0, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}{\langle \texttt{var}\ xl\ ;\ s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}$$

We preferred to also keep the input buffer in the final configuration for two reasons: to avoid having to define a new configuration holding only the state and the output, and to allow the language designer to more easily debug her semantics, in particular to see whether there is any input left unused at the end of the program. One can argue that now, since we have output in our language, the result configuration may actually contain only the output (that is, it can also drop the state). The reader is encouraged to experiment with different final configurations.

Hence, almost everything changed in the original big-step SOS of IMP in order to prepare for the addition of the input/output constructs. All theses necessary changes highlight, again, the lack of modularity of big-step SOS. Once all the changes above are applied, one can easily define the semantics of the input/output constructs as follows:

$$\langle \texttt{read()}, \sigma, i : \omega_{in}\rangle \Downarrow \langle i, \omega_{in}\rangle \qquad \text{(BigStep-Read)}$$

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \omega'_{in}\rangle}{\langle \texttt{print(}\,a\,\texttt{)}, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, i\rangle} \qquad \text{(BigStep-Print)}$$

Type System using Big-Step SOS

The typing policy of the input/output constructs is straightforward, though recall that we decided to only allow to read and to print integers. To type programs using input/output, we therefore add the following typing rules to the already existing typing rules in Figure 3.10:

$$xl \vdash \texttt{read()} : \mathit{int} \qquad \text{(BigStepTypeSystem-Read)}$$

$$\frac{xl \vdash a : \mathit{int}}{xl \vdash \texttt{print(}\,a\,\texttt{)} : \mathit{stmt}} \qquad \text{(BigStepTypeSystem-Print)}$$

Small-Step SOS

Like in big-step SOS, in small-step SOS we also need to change all IMP's configurations in Section 3.3.1 in order to add input/output. In the spirit of making only minimal changes, we modify the configurations holding expressions to also only hold an input buffer, and the configurations holding statements to also hold both an input and an output buffer. Implicitly, all IMP's small-step SOS rules in Section 3.3.2 also need to change. The changes are straightforward, essentially having to just propagate the output through each statement construct, but they are still changes and thus expose, again, the lack of modularity of small-step SOS. Here is, for example, how the small-step SOS rules for division and for sequential composition need to change:

$$\frac{\langle a_1, \sigma, \omega_{in}\rangle \to \langle a_1', \sigma, \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle a_1' \,/\, a_2, \sigma, \omega_{in}'\rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in}\rangle \to \langle a_2', \sigma, \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle a_1 \,/\, a_2', \sigma, \omega_{in}'\rangle}$$

$$\frac{\langle s_1, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s_1', \sigma', \omega_{in}', \omega_{out}'\rangle}{\langle s_1 \;;\; s_2, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s_1' \;;\; s_2, \sigma', \omega_{in}', \omega_{out}'\rangle}$$

The expression configurations do not need to consider an output buffer because, as already discussed, in this language design experiment we assume at each stage only the current feature, without attempting to anticipate other features that will be possibly added in the future, and we attempts to do minimal changes. For example, if functions were to be added to the language later, in which case expressions will also possibly affect the output through function calls, then all the expression configurations and their corresponding small-step SOS rules will need to change again.

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the small-step SOS rule for programs to initialize the output buffer to ϵ as follows:

$$\langle\, \texttt{var}\, xl \;;\; s, \omega_{in}\rangle \to \langle s, (xl \mapsto 0), \omega_{in}, \epsilon\rangle$$

Once all the changes are applied, we can give the small-step SOS of input/output as follows:

$$\langle\, \texttt{read()}, \sigma, i : \omega_{in}\rangle \to \langle i, \sigma, \omega_{in}\rangle \qquad \text{(SmallStep-Read)}$$

$$\frac{\langle a, \sigma, \omega_{in}\rangle \to \langle a', \sigma, \omega_{in}'\rangle}{\langle\, \texttt{print(}\, a\,\texttt{)}, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle\, \texttt{print(}\, a'\,\texttt{)}, \sigma, \omega_{in}', \omega_{out}\rangle} \qquad \text{(SmallStep-Print-Arg)}$$

$$\langle\, \texttt{print(}\, i\,\texttt{)}, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle\, \texttt{skip}\,, \sigma, \omega_{in}, \omega_{out} : i\rangle \qquad \text{(SmallStep-Print)}$$

Denotational Semantics

To accommodate the input and the output buffers, the denotation functions associated to IMP's syntactic categories need to change their types from

$$[\![\_]\!] : \mathit{AExp} \to (\mathit{State} \rightharpoonup \mathit{Int})$$
$$[\![\_]\!] : \mathit{BExp} \to (\mathit{State} \rightharpoonup \mathit{Bool})$$
$$[\![\_]\!] : \mathit{Stmt} \to (\mathit{State} \rightharpoonup \mathit{State})$$
$$[\![\_]\!] : \mathit{Pgm} \to \mathit{State}_\perp$$

to

$$[\![\_]\!] : AExp \to (State \times Buffer \rightharpoonup Int \times Buffer)$$
$$[\![\_]\!] : BExp \to (State \times Buffer \rightharpoonup Bool \times Buffer)$$
$$[\![\_]\!] : Stmt \to (State \times Buffer \rightharpoonup State \times Buffer \times Buffer)$$
$$[\![\_]\!] : Pgm \to (Buffer \rightharpoonup State \times Buffer \times Buffer)$$

We next briefly explain the definitions of the new denotation functions.

The denotations of expressions now take a state and an input buffer and produce a value and a possibly modified input buffer (since `read()` may consume elements from the input buffer). For example, the denotation of division becomes:

$$[\![a_1 \,/\, a_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_1) \;/_{Int}\; 1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2)) & \text{if } 1^{\text{st}}(arg_2) \neq 0 \\ \bot & \text{if } 1^{\text{st}}(arg_2) = 0 \end{cases}$$

where $arg_1 = [\![a_1]\!]\pi$ and $arg_2 = [\![a_2]\!](1^{\text{st}}(\pi), 2^{\text{nd}}(arg_1))$.

The denotations of statements can now produce an output buffer in addition to a modified input buffer (and a state). For example, the denotation of sequential composition becomes:

$$[\![s_1 \;;\; s_2]\!]\pi = (1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2),\; 3^{\text{rd}}(arg_1) : 3^{\text{rd}}(arg_2))$$

where $arg_1 = [\![s_1]\!]\pi$ and $arg_2 = [\![s_2]\!](1^{\text{st}}(arg_1),\; 2^{\text{nd}}(arg_1))$. As another example of statement denotational semantics, the denotational semantics of while loops $\texttt{while}\, b\, \texttt{do}\, s$ remains a fixed-point, but in order to be consistent with the new type of the denotation function for statements, it needs to be the fixed point of a (total) function of the form

$$\mathcal{F} : (State \times Buffer \rightharpoonup State \times Buffer \times Buffer) \to (State \times Buffer \rightharpoonup State \times Buffer \times Buffer)$$

It is not difficult to see that the following definition of $\mathcal{F}$ has the right type and that $\mathcal{F}(\alpha)$ indeed captures the information that is added to α by unrolling the loop once:

$$\mathcal{F}(\alpha)(\pi) = \begin{cases} (1^{\text{st}}(arg_3),\; 2^{nd}(arg_3),\; 3^{\text{rd}}(arg_2) : 3^{\text{rd}}(arg_3)) & \text{if } 1^{\text{st}}(arg_1) = \texttt{true} \\ (1^{\text{st}}(\pi),\; 2^{\text{nd}}(arg_1),\; \epsilon) & \text{if } 1^{\text{st}}(arg_1) = \texttt{false} \\ \bot & \text{if } arg_1 = \bot \end{cases}$$

where $arg_1 = [\![b]\!]\pi$, $arg_2 = [\![s]\!](1^{\text{st}}(\pi),\; 2^{\text{nd}}(arg_1))$, and $arg_3 = \alpha(1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2))$.

Programs now take an input as well; like for big-step SOS, we prefer to also make the remaining input available at the end of the program execution, in addition to the state and the output:

$$[\![\texttt{var}\, xl \;;\; s]\!]\omega = [\![s]\!]((xl \mapsto 0), \omega)$$

Once all the changes on the denotations of the various syntactic categories are applied as discussed above, adding the semantics of the new input/output constructs is immediate:

$$[\![\texttt{read()}]\!]\pi = (head(2^{\text{nd}}(\pi)),\; tail(2^{\text{nd}}(\pi))) \qquad \text{(DENOTATIONAL-READ)}$$

$$[\![\texttt{print(}\, a\, \texttt{)}]\!]\pi = (1^{\text{st}}(\pi),\; 2^{\text{nd}}([\![a]\!]\pi),\; 1^{\text{st}}([\![a]\!]\pi)) \qquad \text{(DENOTATIONAL-PRINT)}$$

Like in the case of IMP's extension with the variable increment expression construct, the denotational semantics of the extension with the input/output constructs would have been more modular if we had adopted a continuation or monadic style from the very beginning.

3.5.3 Adding Abrupt Termination

IMP++ adds both implicit and explicit abrupt program termination. The implicit abrupt termination is given by division by zero, while the explicit abrupt termination is given by a new statement added to the language, `halt`.

For the sake of making a choice and also for demonstration purposes, in both cases of abrupt termination we would like, admittedly subjectively, the resulting configuration to have the same structure as if the program terminated normally; for example, in the case of big-step SOS, we would like the result configuration for statements to be $\langle \sigma \rangle$, where σ is the state when the program was terminated abruptly. For example, we want the programs

```
var m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else halt
```

```
var m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else s := s / (n / m)
```

to yield the result configuration `< m |-> 101 & n |-> 100 & s |-> 5050 >` instead of a special configuration of the form `< halting, m |-> 101 & n |-> 100 & s |-> 5050 >` or similar. Unfortunately, that is not possible in all cases without intrusively modifying the syntax of the IMP language (to catch the exceptional behavior and explicitly discard the additional information), since some operational styles need to make a sharp distinction between a halting configuration and a normal configuration (for propagation reasons). Proponents of those semantic styles may argue that our semantic choice above seems inappropriate, since giving more information in the result configuration, such as "this is a halting configuration", is better for all purposes than giving less information. There are, however, also reasons to always want a normal result configuration upon termination. For example, one may want to include IMP in a larger context, such as in a distributed system, where all the context wants to know about the embedded language is that it takes a statement and produces a state and/or an output; IMP's internal exceptional situations are of no concern to the outer context. There is no absolute better or worse language design, both in what regards syntax and in what regards semantics. Our task here is to make the language designer aware of the subtleties and the limitations of the various semantic approaches.

Big-Step SOS

The lack of modularity of big-step semantics will be, again, emphasized here. Let us first add the semantic definition for the implicit abrupt termination generated by division by zero. Recall that the big-step SOS rule for division was the following:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle i_1 \ /_{Int} \ i_2 \rangle} \text{, where } i_2 \neq 0$$

We keep that unchanged, but we also add the following new rule:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle 0 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-By-Zero)}$$

In the above rule, `error` can be regarded as a special value; alternatively, one can regard $\langle \texttt{error} \rangle$ as a special result configuration.

But what if the evaluation of a_1 or of a_2 in the above rule generates itself an error? If that is the case, then one needs to propagate that error through the division construct:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-Error-Left)}$$

$$\frac{\langle a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-Error-Right)}$$

Note that in case one of a_1 or a_2 generates an error, then the other one is not even evaluated anymore, to faithfully capture the intended meaning of abrupt termination.

Unfortunately, one has to do the same for all the expression language constructs. This way, for each expression construct, one has to add at least as many error-propagation big-step SOS rules as arguments that expression construct takes. Moreover, when the evaluation error reaches a statement, one needs to transform it into a "halting signal". This can be achieved by introducing a new type of result configuration, namely $\langle \texttt{halting}, \sigma \rangle$, and then adding appropriate halting propagation rules for all the statements. For example, the assignment statement needs to be added the new rule

$$\frac{\langle a, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle x \texttt{ := } a, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma \rangle} \qquad \text{(BigStep-Asgn-Halt)}$$

The halting signal needs to be propagated through statement constructs, collecting the appropriate state. For example, the following two rules need to be included for sequential composition, in addition to the existing rule (which stays unchanged):

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma_1 \rangle}{\langle s_1 \texttt{;} s_2, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma_1 \rangle} \qquad \text{(BigStep-Seq-Halt-Left)}$$

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle,\ \langle s_2, \sigma_1 \rangle \Downarrow \langle \texttt{halting}, \sigma_2 \rangle}{\langle s_1 \texttt{ ; } s_2, \sigma \rangle \Downarrow \langle \texttt{ halting }, \sigma_2 \rangle} \qquad \text{(BigStep-Seq-Halt-Right)}$$

In addition to all the halting propagation rules, we also have to define the semantics of the explicit halt statement:

$$\langle \texttt{ halt }, \sigma \rangle \Downarrow \langle \texttt{ halting }, \sigma \rangle \qquad \text{(BigStep-Halt)}$$

Therefore, when using big-step SOS, one has to more than *double* the number of rules in order to support abrupt termination. Indeed, any argument of any language construct can yield the termination signal, so a rule is necessary to propagate that signal through the current language construct. It is hard to imagine anything worse in a language design framework. An unfortunate language designer choosing big-step semantics as her favorite language definition framework will incrementally become very reluctant to add or experiment with any new feature in her language. For example, imagine that one wants to add exceptions and break/continue of loops to IMP++.

Finally, unless one extends the language syntax, there appears to be no way to get rid of the junk result configurations $\langle \texttt{ halting }, \sigma \rangle$ that have been artificially added in order to propagate the

error or the halting signals. For example, one cannot simply add the rule

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \texttt{halting}, \sigma'\rangle}{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

because it may interfere with other rules and thus wrongly hide the halting signal; for example, it can be applied on the second hypothesis of the rule (BigStep-Seq-Halt-Right) above hiding the halting signal and thus wrongly making the normal rule (BigStep-Seq) applicable. While having junk result configurations of the form $\langle \texttt{halting}, \sigma\rangle$ may seem acceptable in our scenario here, perhaps even desirable for debugging reasons, in general one may find it inconvenient to have many types of result configurations; indeed, one would need similar junk configurations for exceptions, for break/continue of loops, for functions return, etc.

Consequently, the halting signal needs to be caught at the top-level of the derivation. Fortunately, IMP provides a top-level syntactic category, *Pgm*, so we can add the following rule which dissolves the potential junk configuration at the top:

$$\frac{\langle s, xl \mapsto 0\rangle \Downarrow \langle \texttt{halting}, \sigma\rangle}{\langle \texttt{var}\ xl\ \texttt{;}\ s\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-Halt)}$$

Now we have silent abrupt termination. Note that `var` now acts as an exception catching and dissolving the abrupt termination signal generated by `halt` or by division-by-zero. If one does not like to use `var` for something which has not been originally intended, then one can add an auxiliary `top` statement or program construct, then reduce the semantics of `var` to that of `top`, and then give `top` an exception-handling-like big-step SOS, as we will do for the MSOS definition of abrupt termination in Section 3.6; see Exercise 75. This latter solution is also more general, because it does not rely on a fortunate earlier decision to have a top-level language construct.

In addition to the lack of modularity due to having to more than double the number of rules in order to add abrupt termination, the inclusion of all these rules can also have a significant impact on performance when one wants to execute the big-step SOS. Indeed, there are now four rules for division, each having the same left-hand side, $\langle a_1 \,/\, a_2, \sigma\rangle$, and some of these rules even sharing some of the hypotheses. That means that any general-purpose proof or rewrite system attempting to execute such a definition will unavoidably face the problem of searching a large space of possibilities in order to find one or all possible reductions.

Type System using Big-Step SOS

The typing policy of abrupt termination is clear: `halt` types to a statement and division-by-zero is ignored. Indeed, one cannot expect that a type checker, or any technique, procedure or algorithm, can detect division by zero in general: division-by-zero, like almost any other runtime property of any Turing-complete programing language, is an undecidable problem. Consequently, it is common to limit typing division to checking that the two expressions have the expected type, integer in our case, which our existing type checker for IMP already does (see Figure 3.10). We therefore only add the following typing rule for `halt`:

$$xl \vdash \texttt{halt} : \mathit{stmt} \qquad \text{(BigStepTypeSystem-Halt)}$$

Small-Step SOS

Small-step SOS turns out to be almost as non-modular as big-step SOS when defining control-intensive constructs like abrupt termination. Like for big-step SOS, we need to invent special configurations to signal steps corresponding to implicit division by zero or to explicit halt statements. In small-step SOS, a single type of such special configurations suffices, namely one of the form $\langle \texttt{halting}, \sigma \rangle$, where σ is the state in which the program was abruptly terminated. However, for uniformity with big-step SOS, we also define two types of special configurations, one for expressions and one for statements; since we decided to carry the state in the right-hand-side configuration of all sequents in our small-step SOS definitions, the only difference between the two types of configurations is their tag, namely $\langle \texttt{error}, \sigma \rangle$ for the former versus $\langle \texttt{halting}, \sigma \rangle$ for the latter.

We can then define the small-step SOS of division by zero as follows (recall that the original SmallStep-Div rule in Figure 3.14 is "$\langle i_1 \texttt{ / } i_2, \sigma \rangle \rightarrow \langle i_1 /_{Int} i_2, \sigma \rangle$ where $i_2 \neq 0$"):

$$\langle i_1 \texttt{ / } 0, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle \qquad \text{(SmallStep-Div-By-Zero)}$$

Like for the big-step SOS extension above, we have to make sure that the halting signal is correctly propagated. Here are, for example, the propagation rules through the division construct:

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle} \qquad \text{(SmallStep-Div-Error-Left)}$$

$$\frac{\langle a_2, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle} \qquad \text{(SmallStep-Div-Error-Right)}$$

The two rules above are given in such a way that the semantics is faithful to the intended *computational granularity* of the defined language feature. Indeed, we want division by zero to take one computational step to be reported as an error, as opposed to as many steps as the depth of the context in which the error has been detected; for example, a configuration containing expression `(3 / 0) / 3` should reduce to a halting configurations in one step, not in two. If one added a special error integer value and replaced the two rules above by

$$\langle \texttt{error} \texttt{ / } a_2, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle$$

$$\langle a_1 \texttt{ / error}, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle$$

then errors would be propagated to the top level of the program in as many small-steps as the depth of the context in which the error was generated; we do not want that.

Like in the big-step SOS above, the implicit expression errors need to propagate through the statements and halt the program. One way to do it is to generate an explicit `halt` statement and then to propagate that `halt` statement through all the statement constructs as if it was a special statement value, until it reaches the top. However, as discussed in the paragraph above, that would generate as many steps as the depth of the evaluation contexts in which the `halt` statement is located, instead of just one step as desired. Alternatively, we can use the same approach to propagate the halting configuration through the statement constructs as we used to propagate it through the expression constructs. More precisely, we add transition rules from expressions to statements, like the one below (a similar one needs to be added for the conditional statement):

$$\frac{\langle a, \sigma \rangle \rightarrow \langle \texttt{error}, \sigma \rangle}{\langle x \texttt{ := } a, \sigma \rangle \rightarrow \langle \texttt{halting}, \sigma \rangle} \qquad \text{(SmallStep-Asgn-Halt)}$$

Once the halting signal due to a division by zero reaches the statement level, it needs to be further propagated through the sequential composition, that is, we need to add the following rule:

$$\frac{\langle s_1, \sigma\rangle \to \langle \texttt{halting}, \sigma\rangle}{\langle s_1 \,\texttt{;}\, s_2, \sigma\rangle \to \langle \texttt{halting}, \sigma\rangle} \qquad \text{(SmallStep-Seq-Halt)}$$

Note that we assumed that a halting step does not change the state (used the same σ in both the left and the right configurations). One can prove by structural induction that, in our simple language scenario, that is indeed the case; alternatively, one could have equivalently used σ' instead of σ in the right-hand configurations in the rule above.

Finally, we can also generate a special halting configuration when a `halt` statement is reached:

$$\langle \texttt{halt}, \sigma\rangle \to \langle \texttt{halting}, \sigma\rangle \qquad \text{(SmallStep-Halt)}$$

At this moment, any abruptly terminated program reduces to a special configuration of the form $\langle \texttt{halting}, \sigma\rangle$. Recall that our plan, however, was to terminate the computation with a normal configuration of the form $\langle \texttt{skip}, \sigma\rangle$, regardless of whether the program terminates normally or abruptly. Like in the big-step SOS above, the naive solution to transform a step producing a halting configuration into a normal step using the rule

$$\frac{\langle s, \sigma\rangle \to \langle \texttt{halting}, \sigma\rangle}{\langle s, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle}$$

does not work. Indeed, consider a situation where the rule (SmallStep-Seq-Halt) above could apply. There is nothing to prevent the naive rule above to interfere and transform the halting premise of (SmallStep-Seq-Halt) into a normal step producing a `skip`, which can be further fed to the conventional rule for sequential composition, (SmallStep-Seq-Arg1) in Figure 3.15, hereby continuing the execution of the program as if no abrupt termination took place.

Supposing that one wants to waste no computational steps as an artifact of the particular small-step SOS approach chosen, there seems to be no immediate way to terminate the program with a normal result configuration of the form $\langle \texttt{skip}, \sigma\rangle$ both when the program terminates abruptly and when it terminates normally. One possibility, also suggested in the case of big-step SOS discussed above and followed in the subsequent MSOS definition for abrupt termination in Section 3.6, is to add an auxiliary `top` language construct. With that, we can change the small-step SOS rule for variable declarations to reduce the top-level program to its body statement wrapped under this `top` construct, and then add corresponding rules to propagate normal steps under the `top` while catching the halting steps and transforming them into normal steps at the top-level. Here are four small-step SOS rules achieving this (the first rule replaces the previous one for variable declarations); see also Exercise 78:

$$\langle \texttt{var}\, xl \,\texttt{;}\, s\rangle \to \langle \texttt{top}\, s, (xl \mapsto 0)\rangle$$

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle \texttt{top}\, s, \sigma\rangle \to \langle \texttt{top}\, s', \sigma'\rangle}$$

$$\langle \texttt{top skip}, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle$$

$$\frac{\langle s, \sigma\rangle \to \langle \texttt{halting}, \sigma\rangle}{\langle \texttt{top}\, s, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle}$$

The rules above are such that no computational step is wasted when a halting signal takes place. Unfortunately, we had to introduce additional syntax (the `top` construct) for the sole purpose of making the semantic definition work. If one is willing to waste a computational step in order to explicitly dissolve the halting configuration replacing it by a normal one, then one can add instead the following simple small-step SOS rule, which is also the approach we are taking in the case of small-step SOS in this section, because it gives us, in our view, the best trade-off between elegance and computational intrusiveness (after all, wasting a step in a deterministic manner may be acceptable in most situations):

$$\langle \texttt{halting}, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle \qquad \text{(SmallStep-Halting)}$$

Denotational Semantics

Since both expressions and statements can abruptly terminate, like in the previous semantics we have to provide a means for the denotation functions for expressions and statements to flag when abrupt termination is intended. This way, the denotation of programs can catch the abrupt termination flag and yield the expected state (recall that we want to see normal termination of programs regardless of whether that happens abruptly or not). Specifically, we change the previous denotation functions

$$\begin{array}{l} [\![\_]\!] : AExp \to (State \rightharpoonup Int) \\ [\![\_]\!] : BExp \to (State \rightharpoonup Bool) \\ [\![\_]\!] : Stmt \to (State \rightharpoonup State) \end{array}$$

into denotation functions of the form

$$\begin{array}{l} [\![\_]\!] : AExp \to (State \rightharpoonup Int \cup \{error\}) \\ [\![\_]\!] : BExp \to (State \rightharpoonup Bool \cup \{error\}) \\ [\![\_]\!] : Stmt \to (State \rightharpoonup State \times \{halting, ok\}) \end{array}$$

as described below. Before we proceed, note that the new denotation functions will still associate partial functions to syntactic categories. While the new semantics will be indeed able now to catch and terminate when a division by zero takes place, it will still not be able to catch non-termination of programs; the denotation of those programs will still stay undefined. Strictly technically speaking, the denotations of expressions will now always be defined, because the only source of expression undefinedness in IMP was division by zero. However, for the same reason it is good practice in small-step SOS to have the same type of configurations both to the left and to the right of the transition arrow ($\to$), it is good practice in denotational semantics to work with domains of partial functions instead of ones of total functions. This way, we wouldn't have to change these later on if we add new expression constructs to the language that yield undefinedness (such as, e.g., functions).

The denotation functions of expressions need to change in order to initiate a "catchable" error when division by zero takes place, and then to propagate it through all the other expression constructs. We only discuss the denotation of division, the other being simpler. The previous denotation function of division $[\![a_1 \,/\, a_2]\!]$ was defined as

$$[\![a_1 \,/\, a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \;/_{Int}\; [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

which is not good enough anymore. To catch and propagate the division-by-zero error, we can modify the denotation of division as follows:

$$[\![a_1 \,/\, a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \;/_{Int}\; [\![a_2]\!]\sigma & \text{if } [\![a_1]\!]\sigma \neq \mathit{error} \text{ and } [\![a_2]\!]\sigma \neq \mathit{error} \text{ and } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_1]\!]\sigma = \bot \\ \mathit{error} & \text{if } [\![a_1]\!]\sigma = \mathit{error} \text{ or } [\![a_2]\!]\sigma = \mathit{error} \text{ or } [\![a_2]\!]\sigma = 0 \end{cases}$$

The second case above is necessary, because we want $[\![a_1 \,/\, a_2]\!]\sigma$ to be undefined, and not *error*, when $[\![a_1]\!]\sigma = \bot$ and $[\![a_2]\!]\sigma = 0$.

Like in the previous semantics, the implicit expression errors need to propagate through the statements and halt the program. The denotation function of statements now returns both a state and a flag. The flag tells whether the state resulted from a normal evaluation or whether it is a halting state that needs to be propagated. Here is the new denotation of assignment:

$$[\![x := a]\!]\sigma = \begin{cases} (\sigma, \mathit{halting}) & \text{if } [\![a]\!]\sigma = \mathit{error} \\ \bot & \text{if } \sigma(x) = \bot \text{ or } [\![a]\!]\sigma = \bot \\ (\sigma[[\![a]\!]\sigma/x], \mathit{ok}) & \text{if otherwise} \end{cases}$$

Above, we chose to halt when $[\![a]\!]\sigma = \mathit{error}$ and $\sigma(x) = \bot$ (the cases are handled in order). The alternative would be to choose undefined instead of halt (see Exercise 81). Our choice to assume that the expression being assigned is always evaluated no matter whether the assigned variable is declared or not, is consistent with the small-step SOS of assignment in IMP: first evaluate the expression being assigned step by step, then write the resulting value to the assigned variable if declared; if undeclared then get stuck (Section 3.3.2). The statement sequential composition construct needs to also properly propagate the halting situation:

$$[\![s_1 \,;\, s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } 2^{\text{nd}}([\![s_1]\!]\sigma) = \mathit{halting} \\ [\![s_2]\!](1^{\text{st}}([\![s_1]\!]\sigma)) & \text{if } 2^{\text{nd}}([\![s_1]\!]\sigma) = \mathit{ok} \\ \bot & \text{if } [\![s_1]\!]\sigma = \bot \end{cases}$$

We also discuss the denotational semantics of while loops $\texttt{while}\, b\, \texttt{do}\, s$. It now needs to be the fixed point of a (total) function of the form

$$\mathcal{F} : (\mathit{State} \rightharpoonup \mathit{State} \times \{\mathit{halting}, \mathit{ok}\}) \to (\mathit{State} \rightharpoonup \mathit{State} \times \{\mathit{halting}, \mathit{ok}\})$$

The following defines such an $\mathcal{F}$ which has the right type and captures the information that is added by unrolling the loop once (the cases are handled in order, from top to bottom):

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \bot & \text{if } [\![b]\!]\sigma = \bot \\ (\sigma, \mathit{halting}) & \text{if } [\![b]\!]\sigma = \mathit{error} \\ (\sigma, \mathit{ok}) & \text{if } [\![b]\!]\sigma = \texttt{false} \\ [\![s]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \text{ and } 2^{\text{nd}}([\![s]\!]\sigma) = \mathit{halting} \\ \alpha(1^{\text{st}}([\![s]\!]\sigma)) & \text{if } [\![b]\!]\sigma = \texttt{true} \text{ and } 2^{\text{nd}}([\![s]\!]\sigma) = \mathit{ok} \end{cases}$$

After we modify the denotation functions of almost all expression and statement constructs as explained above (except for the denotations of variable lookup, and of builtin integers and booleans), we have to also modify the denotation of programs to silently discard the halting signal in case its body statement terminated abruptly (the type of the denotation of programs does not change):

$$[\![\texttt{var}\, xl \,;\, s]\!] = 1^{\text{st}}([\![s]\!](xl \mapsto 0))$$

Finally, we can now also give the denotational semantics of `halt`:

$$[\![\texttt{halt}]\!]\sigma = (\sigma, \mathit{halting})$$

Like for the other semantic extensions in this section, adding the semantics of abrupt termination is easy; the tedious part is to modify the existing semantics to make it aware of abrupt termination.

3.5.4 Adding Dynamic Threads

IMP++ adds threads to IMP, which can be dynamically created and terminated. As in the previous IMP extensions, we keep the syntax and the semantics of threads minimal. Our only syntactic extension is a `spawn` statement construct. The semantics of `spawn` S is that the statement S executes concurrently with the rest of the program, sharing and possibly concurrently modifying the same variables, like threads do. The thread corresponding to S terminates when S terminates and, when that happens, we remove the thread. Like in the previous language extensions, we want programs to terminate normally, no matter whether they make use of threads or not. For example, the programs below create 101 and, respectively, 2 new threads during their execution:

```
var m, n, s ;
n := 100 ;
while (m <= n) do (
  spawn s := s + m ;
  m := m + 1
)
```

```
var x ;
(
   spawn x := x + 1 ;
   spawn x := x + 10
) ;
x := x + 100
```

Yet, we want the result configurations at the end of the execution to look like before in IMP, that is, like `< skip, m |-> 101 & n |-> 100 & s |-> 5050 >` and `< skip, x |-> 111 >`, respectively, in the case of small-step SOS. We grouped the two `spawn` statements in the second program on purpose, to highlight the need for structural equivalences (this will be discussed shortly, in the context of small-step SOS). Recall that the syntax of IMP's sequential composition in Section 3.1 (see Figure 3.1) was deliberately left ambiguous, based on the hypothesis that the semantics of IMP will be given in such a way (left-to-right statement evaluation) that the syntactic ambiguity is irrelevant. Unfortunately, the addition of threads makes the above hard or impossible to achieve modularly in some of the semantic approaches.

Concurrency often implies non-determinism, which is not always desirable. For example, the first program above can also evaluate to a result configuration in which `s` is 0. This happens when the first spawned thread calculates the sum `s + m`, which is 0, but postpones writing it to `s` until all the subsequent 100 new threads complete their execution. Similarly, after the execution of the second program above, `x` can have any of the values 1, 10, 11, 100, 101, 110, 111 (see Exercise 82).

A language designer or semanticist may find it very useful to execute and analyze programs like above in her semantics. Indeed, the existence of certain behaviors or their lack of, may suggest changes in the syntax and the semantics of the language at an early and therefore cheap design stage. For example, one may decide that one's language must be race-free on any variable except some special semaphore variables used specifically for synchronization purposes (this particular decision may be too harsh on implementations, though, which may end up having to rely on complex static analysis front-ends or even ignoring it). We make no such difficult decisions in our simple language here, limiting our goal to the bottom of the spectrum of semantic possibilities: we only aim at giving a semantics that faithfully captures all the program behaviors due to spawning threads.

We make the simplifying assumptions that we have a sequentially consistent memory and that variable read and write operations are atomic and thus unaffected by potential races. For example,

if `x` is 0 in a two-threaded program where one thread is about to write 5 to `x` and the other is about to read `x`, then the only global next steps can be either that the first thread writes 5 to `x` or that the second thread reads 0 as the value of `x`; in other words, we assume that it is impossible for the second thread to read 5 or any other value except 0 as the next small-step step in the program.

Big-Step SOS

Big-step SOS and denotational semantics are the semantical approaches which are the most affected by concurrency extensions of the base language. That is because their holistic view of computation makes it hard or impossible to capture the fine-grained execution steps and behavior interleavings that are inherent to concurrent program executions. Consider, for example, a statement of the form $\texttt{spawn}\, S_1 \texttt{ ; } S_2$ in some program state σ. The only thing we can do in big-step SOS is to evaluate S_1 and S_2 in some appropriate states and then to combine their resulting states into a result state. We do not have much room for imagination here: we either evaluate S_1 in state σ and then S_2 in the resulting state, or evaluate S_2 in state σ and then S_1 in the resulting state. The big-step SOS rule for sequential composition already implies the former case provided that $\texttt{spawn}\, S$ can evaluate to whatever state S evaluates to, which is true and needs to be considered anyway. Thus, we can formalize the above into the following two big-step SOS rules, which can be regarded as a rather desperate attempt to use big-step SOS for defining concurrency:

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle\, \texttt{spawn}\, s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-Spawn-Arg)}$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle \qquad \langle s_1, \sigma_2\rangle \Downarrow \langle \sigma_1\rangle}{\langle\, \texttt{spawn}\, s_1 \texttt{ ; } s_2, \sigma\rangle \Downarrow \langle \sigma_1\rangle} \qquad \text{(BigStep-Spawn-Wait)}$$

As expected, the two big-step SOS rules above capture only a limited number of the possible concurrent behaviors even for small and simple programs like the ones discussed above. One may try to change the entire big-step SOS definition of IMP to collect in result configurations all possible ways in which the corresponding fragments of program can evaluate. However, in spite of its non-modularity, there seems to be no easy way to combine, for example, the behaviors of S_1 and of S_2 into the behaviors of $\texttt{spawn}\, S_1 \texttt{ ; } S_2$.

Type System using Big-Step SOS

From a typing perspective, `spawn` is nothing but a language construct expecting a statement as argument and producing a statement as result. To type programs using `spawn` statements we therefore add the following typing rule to the already existing typing rules in Figure 3.10:

$$\frac{xl \vdash s : stmt}{xl \vdash \texttt{spawn}\, s : stmt} \qquad \text{(BigStepTypeSystem-Spawn)}$$

Small-Step SOS

Small-step semantics are more appropriate for concurrency, because they allow a finer-grain view of computation. For example, they allow to say that the next computational step of a statement of the form $\texttt{spawn}\, S_1 \texttt{ ; } S_2$ comes either from S_1 or from S_2 (which is different from saying that either

S_1 or S_2 is evaluated next all the way through, like in big-step SOS). Since $\texttt{spawn}\, S_1$ is already permitted to advance by the small-step SOS rule for sequential composition, the following three small-step SOS rules achieve the desired behavioral non-determinism caused by concurrent threads:

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle\, \texttt{spawn}\, s, \sigma\rangle \to \langle\, \texttt{spawn}\, s', \sigma'\rangle} \qquad \text{(SmallStep-Spawn-Arg)}$$

$$\langle\, \texttt{spawn skip}\,, \sigma\rangle \to \langle\, \texttt{skip}\,, \sigma\rangle \qquad \text{(SmallStep-Spawn-Skip)}$$

$$\frac{\langle s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}{\langle\, \texttt{spawn}\, s_1\, \texttt{;}\, s_2, \sigma\rangle \to \langle\, \texttt{spawn}\, s_1\, \texttt{;}\, s_2', \sigma'\rangle} \qquad \text{(SmallStep-Spawn-Wait)}$$

The rule (SmallStep-Spawn-Skip) cleans up terminated threads.

Unfortunately, the three rules above are not sufficient to capture all the intended behaviors. Consider, for example, the second program at the beginning of Section 3.5.4. That program was intended to have seven different behaviors with respect to the final value of x. Our current small-step SOS misses two of those behaviors, namely those in which x results in 1 and 100, respectively.

In order for the program to terminate with $x = 1$, it needs to start the first new thread, calculate the sum `x + 1` (which is 1), then delay writing it back to x until after the second and the main threads do their writes of x. However, in order for the main thread to be allowed to execute its assignment statement, the two grouped `spawn` statements need to either terminate and become `skip` so that the rule (SmallStep-Seq-Skip) (see Figure 3.15) applies, or to reduce to only one `spawn` statement so that the rule (SmallStep-Spawn-Wait) above applies. Indeed, these are the only two rules which allow access to the second statement in a sequential composition. The first case is not possible, because, as explained, the first newly created thread cannot be terminated. In order for the second case to happen, since the first `spawn` statement cannot terminate, the only possibility is for the second `spawn` statement to be executed all the way through (which is indeed possible, thanks to the rules (SmallStep-Seq-Arg1) in Figure3.15 and (SmallStep-Spawn-Wait) above) and then eliminated. To achieve this elimination, we may think of adding a new rule, which appears to be so natural that one may even wonder "how did we miss it in our list above?":

$$\langle\, \texttt{spawn}\, s\, \texttt{;}\ \texttt{skip}\,, \sigma\rangle \to \langle\, \texttt{spawn}\, s, \sigma\rangle$$

This rule turns out to be insufficient and, once we fix the semantics properly, it will actually become unnecessary. Nevertheless, once we add this rule, the resulting small-step SOS can also produce the behavior in which $x = 1$ at the end of the execution of the second program at the beginning of Section 3.5.4. However, it is still insufficient to produce the behavior in which $x = 100$.

In order to produce a behavior in which $x = 100$ when executing the second program, the main thread should first execute its `x + 100` step (which evaluates to 100), then let the two child threads do their writes to x, and then write the 100 to x. We have, unfortunately, no rule that allows computations within s_2 in a sequential composition $s_1\, \texttt{;}\, s_2$ where s_1 is different from `skip` or a `spawn` statement, as it is our case here. What we want is some generalization of the rule (SmallStep-Spawn-Wait) above which allows computations in s_2 whenever it is preceded by a sequence of spawns. On paper definitions, one can do that rather informally by means of some informal side condition saying so. If one needs to be formal, which is a must when one needs to execute the resulting language definitions as we do here, one can define a special sequent saying that a statement only spawns new threads and does nothing else (in the same spirit as the the $C\surd$

sequents in Exercise 48), and then use it to generalize the rule (SmallStep-Spawn-Wait) above. However, that would be a rather particular and potentially non-modular (what if later on we add agents or other mechanisms for concurrency?) solution.

Our general solution is to instead enforce the sequential composition of IMP to be structurally associative, using the following structural identity:

$$(s_1 \texttt{;} s_2) \texttt{;} s_3 \equiv s_1 \texttt{;} (s_2 \texttt{;} s_3) \qquad \text{(SmallStep-Seq-Assoc)}$$

That means that the small-step SOS reduction rules now apply *modulo* the associativity of sequential composition, that is, that it suffices to find a structurally equivalent representative of a syntactic term which allows a small-step SOS rule to apply. In our case, the original program is structurally equivalent to one whose first statement is the first `spawn` and whose second statement is the sequential composition of the second `spawn` and the assignment of the main thread, and that structurally equivalent program allows all seven desired behaviors, so the original program also allows them. It is important to understand that we cannot avoid enforcing associativity (or, alternatively, the more expensive solution discussed above) by simply parsing the original program so that we start with a right-associative arrangement of the sequentially composed statements. The problem is that right-associativity may be destroyed as the program executes, for example when applying the true/false rules for the `if` statement, so it needs to be dynamically enforced.

Structural identities are not easy to execute and/or implement, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Since in our particular case we only need the fully right-associative representative of each sequential composition, we can even replace the structural identity above by a small-step SOS rule. The problem with this is that the intended computational granularity of the language is significantly modified; for example, the application of a true/false rule for the conditional statement may trigger as many such artificial rearrangement steps as statements in the chosen branch, to an extent that such rearrangement steps could dominate the total number of steps seen in some computations.

Denotational Semantics

As already mentioned when we discussed the big-step SOP of `spawn` above, big-step SOS and denotational semantics are the semantic approaches which are the most affected by the addition of concurrency to IMP. While big-step SOS was somewhat able to capture some of the non-determinism due to concurrency, unfortunately, denotational semantics cannot do even that easily. The notes on denotational semantics in Section 3.4.3 mention the use of powerdomains and resumptions when giving concurrent semantics to languages. These are complex denotational semantics topics, which are not easy to use even by experts. Moreover, they yield semantics which are either non-executable at all or very slow. Since the main emphasis of this book is on operational semantics, we do not discuss these advanced topics in this book. Instead, we simply dissolve the `spawn` statements, so we can still execute IMP++ programs using denotational semantics:

$$[\![\texttt{spawn}\, s]\!] = [\![s]\!]$$

Of course, spawning threads is completely useless with our denotational semantics here.

3.5.5 Adding Local Variables

Blocks with local variable declarations are common to many imperative, object oriented and functional languages. In IMP++ we follow the common notation where a block is a sequence of

statements surrounded with curly brackets. Variables can be declared anywhere inside a block, their *scope* being the rest of the block (whatever follows the declaration); in other words, a declared variable is not visible before its declaration or outside the block declaring it. A declaration of a variable that appears in the scope of another declaration of the same variable is said to *shadow* the original one. For example, the values of `y` and `z` are 1 and 2, respectively, right before the end of the following two IMP++ blocks (none of the variables are visible outside the given blocks):

```
{  var x,y,z ;
   x := 1 ;
   y := x ;
   var x ;
   x := 2 ;
   z := x  }
```

```
{  var x,y,z ;
   x := 1 ;
   {  var x ;
      x := 2 ;
      z := x  } ;
   y := x  }
```

As already explained in the preamble of Section 3.5, the introduction of blocks and local variables suggests some syntactic and semantic simplifications in the already existing definition of IMP. For example, since local variable declarations generalize the original global variable declarations of IMP, there is no need for the original global declarations. Thus, programs can be just statements. Therefore, we remove the top-level variable declaration and add the following new syntax:

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \{\} \\ & | & \{\ \mathit{Stmt}\ \} \\ & | & \texttt{var}\ \mathbf{List\{}\mathit{Id}\mathbf{\}} \\ \mathit{Pgm} & ::= & \mathit{Stmt} \end{array}$$

In each of the semantics, we assume that all the previous rules referring to global variable declarations are removed. Moreover, for semantic clarity, we assume that variable declarations can only appear in blocks (a hypothetical parser can reject those programs which do not conform).

An immediate consequence of this language extension and the conventions above is that programs now evaluate to empty states. Indeed, since the initial state in which a program is evaluated is empty and since variable declarations are local to the blocks in which they occur, the state obtained after evaluating a program is empty. This makes it somewhat difficult to test this IMP extension. To overcome this problem, one can either add an output statement to the language like in Section 3.5.2 (we will do this in Section 3.5.6), or one can manually initialize the state with some "global" variables and then use those variables undeclared in the program.

It would be quite tedious to give semantics directly to the syntactic constructs above. Instead, we are going to propose another construct which is quite common and easy to give semantics to in each of the semantic approaches, and then statically translate the constructs above into the new construct. The new construct has the following syntax:

$$\mathit{Stmt} \ ::= \ \texttt{let}\ \mathit{Id} = \mathit{AExp}\ \texttt{in}\ \mathit{Stmt}$$

Its semantics is as expected: the arithmetic expression is first evaluated to an integer, then the declared variable is bound to that integer possibly shadowing an already existing binding of the same variable, then the statement is evaluated in the new state, and finally the environment before the execution of the `let` is recovered. The latter step is executed by replacing the value of the variable after the execution of the statement with whatever it was before the `let`, possibly undefined. All the other side effects generated by the statement are kept.

Here we propose a simple set of macros which automatically desugar any program using the block and local variable constructs into a program containing only `let` and the existing IMP constructs:

$$\begin{array}{rcl}
(s_1 \,;\, s_2) \,;\, s_3 & = & s_1 \,;\, (s_2 \,;\, s_3) \\
\texttt{var}\, xl, x \,;\, s & = & \texttt{var}\, xl \,;\, \texttt{let}\, x = 0\, \texttt{in}\, s \\
\texttt{var}\, \cdot \,;\, s & = & s \\
s \,;\, \texttt{var}\, xl & = & s \\
\{\, \texttt{var}\, xl \,\} & = & \texttt{skip} \\
\{\, s \,\} & = & s \\
\{\} & = & \texttt{skip}
\end{array}$$

The macros above are expected to be iteratively applied in order, from top to bottom, until no macro can be applied anymore. When that happens, there will be no block or variable declaration left; all of these would have been systematically replaced by the `let` construct. The first macro enforces right-associativity of sequential composition. This way, any non-terminal variable declaration (i.e., one which is not the last statement in a block) will be followed, via a sequential composition, by the remainder of the block. The second and the third macros iteratively replace each non-terminal variable declaration by a corresponding `let` statement, while the fourth and the fifth eliminate the remaining (and useless) terminal variable declarations. Finally, the sixth and the seventh macros eliminate the blocks, which are unnecessary now. From here on we assume that these syntactic desugaring macros are applied statically, before any of the semantic rules is applied; this way, the subsequent semantics will only be concerned with giving semantics to the `let` construct.

Big-Step SOS

The big-step SOS rule of `let` follows quite closely its informal description above:

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle \qquad \langle s, \sigma[i/x]\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{let}\, x = a\, \texttt{in}\, s, \sigma\rangle \Downarrow \langle \sigma'[\sigma(x)/x]\rangle} \qquad \text{(BigStep-Let)}$$

In words, the arithmetic expression a is first evaluated to some integer i. Then the statement s is evaluated in state $\sigma[i/x]$, resulting in a state σ'. Then we return $\sigma'[\sigma(x)/x]$ as the result of the `let` statement, that is, the state σ' in which we update the value of x to whatever x was bound to originally in σ. We *cannot* return σ' as the result of the `let`, because σ' binds x to some value which is likely different from what x was bound to in σ (note that s is allowed to assign to x, although that is not the main problem here). If σ is undefined in x, that is, if $\sigma(x) = \bot$, then $\sigma'[\sigma(x)/x]$ is also undefined in x: indeed, recall from Section 2.3.2 that $\sigma'[\bot/x]$ "undefines" σ' in x.

Since programs are now just statements, their big-step SOS simply reduces to that of statements:

$$\frac{\langle s, \cdot\rangle \Downarrow \langle \sigma\rangle}{\langle s\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-Pgm)}$$

Hence, programs are regarded as statements that execute in the empty state. However, since variable accesses in IMP require the variable to be declared and since all variable declarations are translated into `let` statements, which recover the state in the variable they bind after their execution, we can conclude that σ will always be empty whenever a sequent of the form $\langle s\rangle \Downarrow \langle \sigma\rangle$ is derivable using the big-step SOS above. The rule above is, therefore, not very practical. All it tells us is that if $\langle s\rangle \Downarrow \langle \sigma\rangle$ is derivable then s is a well-formed program which terminates. The idea of reducing the semantics of statement-programs to that of statements in an initial state like above is general though, and it becomes practical when we add other features to the language (see Section 3.5.6).

Type System using Big-Step SOS

Following the intuitions above, to type programs using `let` statements we add the following typing rules to the already existing typing rules in Figure 3.10:

$$\frac{xl \vdash a : int \qquad xl, x \vdash s : stmt}{xl \vdash \texttt{let}\ x = a\ \texttt{in}\ s : stmt} \qquad \text{(BigStepTypeSystem-Let)}$$

$$\frac{\cdot \vdash s : stmt}{\vdash s : pgm} \qquad \text{(BigStepTypeSystem-Pgm)}$$

Small-Step SOS

The small-step SOS of $\texttt{let}\ x = a\ \texttt{in}\ s$ can be described in words as follows: first evaluate a stepwise, until it becomes some integer; then evaluate s stepwise in a state originally binding x to the integer to which a evaluates, but making sure that the value bound to x is properly updated during each step in the evaluation of s and it is properly recovered after each step to whatever it was in the environment outside the `let` (so other potentially interleaved rules taking place outside the `let` see a consistent state); finally, dissolve the `let` when its enclosed statement becomes `skip`. All these can be achieved with the following three rules, without having to change anything in the already existing small-step SOS of IMP:

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle \texttt{let}\ x = a\ \texttt{in}\ s, \sigma\rangle \to \langle \texttt{let}\ x = a'\ \texttt{in}\ s, \sigma\rangle} \qquad \text{(SmallStep-Let-Exp)}$$

$$\frac{\langle s, \sigma[i/x]\rangle \to \langle s', \sigma'\rangle}{\langle \texttt{let}\ x = i\ \texttt{in}\ s, \sigma\rangle \to \langle \texttt{let}\ x = \sigma'(x)\ \texttt{in}\ s', \sigma'[\sigma(x)/x]\rangle} \qquad \text{(SmallStep-Let-Stmt)}$$

$$\langle \texttt{let}\ x = i\ \texttt{in}\ \texttt{skip}, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle \qquad \text{(SmallStep-Let-Done)}$$

Note that if x was undeclared before the `let` then so it stays after each application of the rule (SmallStep-Let-Stmt), because $\sigma'[\bot/x]$ "undefines" σ' in x (see Section 2.3.2).

Like in big-step SOS, the semantics of programs (which are now statements) reduces to that of statements. One simple way to achieve that in small-step SOS is to add a rule $\langle s\rangle \to \langle s, \cdot\rangle$, in the same spirit as the small-step SOS of IMP in Section 3.3.2. However, like in Exercise 51, one could argue that this approach is wasteful, since one does not want to spend a step only to initialize the empty state (this can be regarded as poor style). For demonstration purposes and for the sake of a semantic variation, we next show a non-wasteful small-step SOS rule of programs:

$$\frac{\langle s, \cdot\rangle \to \langle s', \sigma\rangle}{\langle s\rangle \to \langle s', \sigma\rangle} \qquad \text{(SmallStep-Pgm)}$$

One could still argue that the rule above is not perfect, because the configuration $\langle \texttt{skip}\rangle$ is frozen; thus, while any other (terminating) program eventually reduces to a configuration of the form $\langle \texttt{skip}, \cdot\rangle$, `skip` itself does not. To address this non-uniformity problem, one can add a rule $\langle \texttt{skip}\rangle \to \langle \texttt{skip}, \cdot\rangle$; this wastes a step, indeed, but this case when the entire program is just `skip` is expected to be very uncommon. A conceptually cleaner alternative is to replace the rule (SmallStep-Pgm) above with a structural identity $\langle s\rangle \equiv \langle s, \cdot\rangle$ identifying each program

configuration with a statement configuration holding an empty state. This can be easily achieved in Maude using an equation, but it can be harder to achieve in other rewrite systems providing support and semantics only for rewrite rules but not for equations.

Denotational Semantics

The denotational semantics of the `let` construct is very compact and elegant:

$$[\![\texttt{let}\ x\texttt{=}a\ \texttt{in}\ s]\!]\sigma = ([\![s]\!](\sigma[[\![a]\!]\sigma/x]))[\sigma(x)/x] \qquad \text{(DENOTATIONAL-LET)}$$

Like in the other semantics above, this works because $\sigma'[\bot/x]$ "undefines" σ' in x (see Section 2.3.2).

3.5.6 Putting Them All Together

We next further analyze the modularity of the various semantic approaches by defining the IMP++ language, which puts together all the language features discussed so far. Recall from the preamble of Section 3.5 that, syntactically, IMP++ removes from IMP the global variable declarations and adds the following constructs:

$$\begin{array}{rcl} \mathit{AExp} & ::= & \texttt{++}\,\mathit{Id} \mid \texttt{read()} \\ \mathit{Stmt} & ::= & \texttt{print(}\,\mathit{AExp}\,\texttt{)} \\ & \mid & \texttt{halt} \\ & \mid & \texttt{spawn}\,\mathit{Stmt} \\ & \mid & \{\} \mid \{\,\mathit{Stmt}\,\} \mid \texttt{var}\ \mathbf{List\{}\mathit{Id}\mathbf{\}} \\ \mathit{Pgm} & ::= & \mathit{Stmt} \end{array}$$

We consider the semantics of these constructs adopted in the previous sections.

To make the design of IMP++ more permissive in what regards its possible implementations, we shall opt for maximum non-determinism whenever such design choices can be made. For example, in the case of division expressions a_1 / a_2, we want to capture all possible behaviors (recall that division is non-deterministic) due to the possibly interleaved evaluations of a_1 and a_2, including all possible abruptly terminated behaviors generated when a_2 evaluates to 0. In particular, we want to also capture those behaviors where a_1 is not completely evaluated. The rationale for this language design decision is that we want to allow maximum flexibility to implementations of IMP++; for example, some implementations may choose to evaluate a_1 and a_2 in two concurrent threads and to stop with abrupt termination as soon as the thread evaluating a_2 yields 0.

Somewhat surprisingly, when adding several new features together to a language, it is not always sufficient to simply apply all the global, non-modular changes that are required for each feature in isolation. We sometimes have to additionally consider the semantic implications of the various combinations of features. For example, the addition of side-effects in combination with division-by-zero abrupt termination requires the addition of new rules in order to catch specific new behaviors due to this particular combination. Indeed, the evaluation of a_1 / a_2, for example, may abruptly terminate with the current state precisely when a_2 is evaluated to zero, but it can also terminate with the state obtained after evaluating a_1 or parts of a_1, as discussed above.

Also, a language design decision needs to be made in what regards the state of abruptly terminated programs. One option is to simply enclose the local state when the abrupt termination flag was issued. This option is particularly useful for debugging. However, as argued in Section 3.5.3, we want abruptly terminated programs to behave the same way as the normally terminated programs.

Since normally terminated programs now empty the state after their execution, we will give the semantics of abruptly terminated programs to also empty the state. Whenever easily possible, we will give the semantics of abruptly terminated statements to return after their evaluation a state binding precisely the same variables as before their evaluation.

A design decision also needs to be made in what regards the interaction between abrupt termination and threads. We (subjectively) choose that abrupt termination applies to the entire program, no matter whether it is issued by the main program or by a spawned thread. An alternative would be that abrupt termination only applies to the spawned thread if issued by a spawned thread, or to the entire program if issued by the main program. Yet another alternative is to consider the main program as an ordinary thread, and an abrupt termination issued by the main program to only stop that thread, allowing the other spawned threads to continue their executions.

Finally, there is an interesting aspect regarding the interaction between blocks and threads. In conventional programming languages, spawned threads continue to execute concurrently with the rest of the program regardless of whether the language construct which generated them completed its execution or not. For example, if function $f()$ spawns a thread and then immediately returns 1, then the expression $f() + f()$ evaluates to 2 and the two spawned threads continue to execute concurrently with the rest of the program. We do not have functions in IMP++, but we still want the spawned threads to continue to execute concurrently with the rest of the program even after the completion of the block within which the spawn statements were executed. For example, we would like the IMP++ program (its `let`-desugared variant is shown to the right)

```
{
   var x ;
   {
      var y ;
      spawn x := x + 1
   } ;
   spawn x := x + 10 ;
   print(x)
}
```

```
let x = 0 in (
   let y = 0 in
      spawn x := x + 1 ;
   spawn x := x + 10 ;
   print(x)
   )
```

to manifest four behaviors, where x is 0, 1, 10, and 11, and not only two (where x is 1, 11) as it would be the case if the first spawn statement were not allowed to transcend its surrounding block.

Below we discuss, for each semantic approach, the changes that we have to apply to the semantics of IMP in order to extend it to IMP++, highlighting changes that cannot be mechanically derived from the changes required by each of IMP++'s features when considered in isolation.

Big-Step SOS

To accommodate the side effects generated by variable increment on the state and by `read()` on the input buffer, and the possible abrupt termination generated by division-by-zero, the arithmetic expression sequents need to change from $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ to $\langle a, \sigma, \omega\rangle \Downarrow \langle i, \sigma', \omega'\rangle$ for normal termination and to $\langle a, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma', \omega'\rangle$ for abrupt termination, and similarly for boolean expressions, where σ, ω and σ',ω' are the states and input buffers before and after the evaluation of a, respectively. Also, the original elegant big-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, the elegant IMP big-step SOS rule for division in Section 3.2.2, namely

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \Downarrow \langle i_1 \ /_{Int} \ i_2\rangle} \quad \text{if } i_2 \neq 0$$

changes into the following six rules:

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle i_1, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma_1, \omega_1\rangle \Downarrow \langle i_2, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2, \omega_2\rangle} \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle i_1, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma_1, \omega_1\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}$$

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1\rangle}$$

$$\frac{\langle a_1, \sigma_2, \omega_2\rangle \Downarrow \langle i_1, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega\rangle \Downarrow \langle i_2, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1, \omega_1\rangle} \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma_2, \omega_2\rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega\rangle \Downarrow \langle i_2, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1\rangle}$$

$$\frac{\langle a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}$$

Like for the individual features, rules like the above attempt to capture the intended nondeterministic evaluation strategy of the arithmetic operators, but they end up capturing only the non-deterministic choice semantics. In the case of division, we also have to add the rule for abrupt termination in the case of a division-by-zero, like the rule (BigStep-Div-By-Zero) in Section 3.5.3. However, since we want to capture all the non-deterministic behaviors that big-step SOS can detect, we actually need three such rules:

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle i_1, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma_1, \omega_1\rangle \Downarrow \langle 0, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}$$

$$\frac{\langle a_1, \sigma_2, \omega_2\rangle \Downarrow \langle i_1, \sigma_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega\rangle \Downarrow \langle 0, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1\rangle}$$

$$\frac{\langle a_2, \sigma, \omega\rangle \Downarrow \langle 0, \sigma_2, \omega_2\rangle}{\langle a_1 \mathbin{/} a_2, \sigma, \omega\rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2\rangle}$$

The big-step SOS sequents for statements $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ also need to change, to hold both the input/output buffers and the termination flag. We use sequents of the form $\langle s, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma', \omega'_{in}, \omega_{out}\rangle$ for the case when s terminates normally and sequents of the form $\langle s, \sigma, \omega_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma', \omega'_{in}, \omega_{out}\rangle$ for the case when s terminates abruptly; here $\omega_{in}, \omega'_{in} \in \mathit{Buffer}$ are the input buffers before and, respectively, after the evaluation of statement s, and $\omega_{out} \in \mathit{Buffer}$ is the output produced during the evaluation of s. All the big-step SOS rules for statements need to change to accommodate the new sequents, making sure that side effects and abrupt termination are properly propagated, like we did in Sections 3.5.1 and 3.5.3 (but for the new configurations). We also have to include big-step SOS rules for input/output like those in Section 3.5.2, for local variable declarations like those in

Section 3.5.5, and for dynamic threads like those in Section 3.5.4, but also modified to work with the new configurations and to propagate abrupt termination. Recall from the preamble of this section that we want abruptly terminated programs to terminate similarly to the normal programs, that is, with an empty state in the configuration. This can be easily achieved in the rule for programs by simply emptying the state when the program statement terminates abruptly. However, in the case of big-step SOS it is relatively easy to ensure a stronger property, namely that each statement leaves the state in a consistent shape after its evaluation, no matter whether that is abruptly terminated or not. All we have to do is to also clean up the state when the halting signal is propagated through the `let` construct. For clarity, we show both big-step SOS rules for `let`:

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \sigma', \omega'_{in}\rangle \qquad \langle s, \sigma'[i/x], \omega'_{in}\rangle \Downarrow \langle \sigma'', \omega''_{in}, \omega_{out}\rangle}{\langle \texttt{let}\ x = a\ \texttt{in}\ s, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma''[\sigma'(x)/x], \omega''_{in}, \omega_{out}\rangle}$$

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \sigma', \omega'_{in}\rangle \qquad \langle s, \sigma'[i/x], \omega'_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma'', \omega''_{in}, \omega_{out}\rangle}{\langle \texttt{let}\ x = a\ \texttt{in}\ s, \sigma, \omega_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma''[\sigma'(x)/x], \omega''_{in}, \omega_{out}\rangle}$$

Recall from Section 2.3.2 that $\sigma''[\perp/x]$ "undefines" σ'' in x.

Finally, the big-step SOS sequents and rules for programs also have to change, to take into account the fact that programs are now just statements like in Section 3.5.5, that they take an input and that they yield both the unconsumed input and an output like in Section 3.5.2, and that programs manifest normal termination behavior no matter whether their corresponding statement terminates normally or not:

$$\frac{\langle s, xl \mapsto 0, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}{\langle s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}$$

$$\frac{\langle s, xl \mapsto 0, \omega_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma, \omega'_{in}, \omega_{out}\rangle}{\langle s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}$$

Unfortunately, as seen above, all the configurations, sequents and rules of IMP extended with any one of the features of IMP++ had to change again when we added all the features together. This highlights, again, the poor modularity of big-step SOS. But, even accepting the poor modularity of big-step SOS, do we at least get all the behaviors expressible in a big-step SOS style by simply putting together and adjusting accordingly all the rules of the individual features? Unfortunately, not. Recall from the preamble of Section 3.5.6 that we want spawned threads to execute concurrently with the rest of the program also after their surrounding blocks complete. In other words, we would like to also capture behaviors of, e.g., $(\texttt{let}\ x = a\ \texttt{in}\ \texttt{spawn}\ s_1)\ \texttt{;}\ s_2$, where a is first evaluated, then s_2, and then s_1. We can easily add a big-step SOS rule to capture this situation, but is that enough? Of course not, because there are many other similar patters in which we would like to allow the evaluation of s_2 before the preceding statement completes its evaluation. For example, one can replace $\texttt{spawn}\ s_1$ above by another `let` holding a spawn statement, or by $\texttt{spawn}\ s'_1\ \texttt{;}\ \texttt{spawn}\ s'_2$, or by combinations of such patterns. A tenacious reader could probably find some complicated way to allow all these behaviors. However, it is fair to say that big-step SOS has not been conceived to deal with concurrent languages, and can only partially deal with non-determinism.

Type System using Big-Step SOS

The IMP++ type system is quite simple and modular. We simply put together all the typing rules of the individual language features discussed in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4, and 3.5.5.

Small-Step SOS

It is conceptually easy, though not entirely mechanical, to combine the ideas and changes to the original IMP small-step SOS discussed in Sections 3.5.1, 3.5.2, and 3.5.3, in order derive the small-step SOS rules of IMP++.

The arithmetic expression sequents need to change from $\langle a, \sigma \rangle \to \langle a', \sigma' \rangle$ to $\langle a, \sigma, \omega \rangle \to \langle a', \sigma', \omega' \rangle$ for normal steps and to $\langle a, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma', \omega' \rangle$ for halting steps, and similarly for boolean expressions, where σ, ω and σ',ω' are the states and input buffers before and after the small-step applied to a, respectively. Also, the original small-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, here are the new rules for division:

$$\frac{\langle a_1, \sigma, \omega_{in} \rangle \to \langle a_1', \sigma', \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle a_1' \,/\, a_2, \sigma', \omega_{in}' \rangle}$$

$$\frac{\langle a_1, \sigma, \omega_{in} \rangle \to \langle \texttt{error}, \sigma', \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle \texttt{error}, \sigma', \omega_{in}' \rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in} \rangle \to \langle a_2', \sigma', \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle a_1 \,/\, a_2', \sigma', \omega_{in}' \rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in} \rangle \to \langle \texttt{error}, \sigma', \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle \texttt{error}, \sigma', \omega_{in}' \rangle}$$

$$\langle i_1 \,/\, i_2, \sigma, \omega_{in} \rangle \to \langle i_1 \,/_{Int}\, i_2, \sigma', \omega_{in}' \rangle \quad \text{if } i_2 \neq 0$$

$$\langle a_1 \;/\; 0, \sigma, \omega_{in} \rangle \to \langle \texttt{error}\,, \sigma, \omega_{in} \rangle$$

Note that the last rule above does not require the full evaluation of a_1 in order to flag abrupt termination. This aspect was irrelevant when we added abrupt termination in isolation to IMP in Section 3.5.3, because expressions did not have side effects there. However, since expressions can now modify both the state (via variable increment) and the input buffer (via `read()`), the rule above captures more behaviors than a rule replacing a_1 by an integer i_1, which would be obtained by mechanically translating the corresponding rule from Section 3.5.3.

The small-step SOS rules of statements and programs straightforwardly result from those of the individual features discussed in Sections 3.5.1, 3.5.2, and 3.5.3. Sequents $\langle s, \sigma \rangle \to \langle s', \sigma' \rangle$ need to change into sequents $\langle s, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle s', \sigma', \omega_{in}', \omega_{out} \rangle$ and $\langle s, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle \texttt{halting}, \sigma', \omega_{in}', \omega_{out} \rangle$ for for normal and for halting steps, respectively, where σ, ω_{in}, ω_{out} and σ', ω_{in}', ω_{out}' are the states, input buffers and output buffers before and after the small-step applied to s, respectively. The only rule that deviates from the expected pattern is the rule that propagates the halting signal through the `let` construct. Like in the case of big-step SOS discussed above, we can do it in such a way that the state is consistently cleaned up after each `let`, regardless of whether its body statement terminated

abruptly or not. Here are all three small-step SOS rules for `let`:

$$\frac{\langle s, \sigma[i/x], \omega_{in}, \omega_{out}\rangle \to \langle s', \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{let}\, x = \sigma'(x)\, \texttt{in}\, s', \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out}\rangle}$$

$$\langle \texttt{let}\, x = i\, \texttt{in}\, \texttt{skip}, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{skip}, \sigma, \omega_{in}, \omega_{out}\rangle$$

$$\frac{\langle s, \sigma[i/x], \omega_{in}, \omega_{out}\rangle \to \langle \texttt{halting}, \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{halting}, \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out}\rangle}$$

Recall again from Section 2.3.2 that $\sigma'[\bot/x]$ "undefines" σ' in x.

Even though strictly speaking all the small-step SOS rules above are different from their corresponding rules in the IMP extension introducing only the feature they define, we claim that they are quite mechanically derivable. In fact, MSOS (see Section 3.6) formalizes and mechanizes their derivation. The main question is then whether these new small-step SOS rules indeed capture the intended semantics of IMP++. Unfortunately, like in the case of big-step SOS, they fail to capture the intended semantics of `spawn`. Indeed, since the `let` construct does not dissolve itself until its body statement becomes `skip`, statements of the form $(\texttt{let}\, x = i\, \texttt{in}\, \texttt{spawn}\, s_1)\, \texttt{;}\, s_2$ will never allow reductions in s_2, thus limiting the concurrency of spawn statements to their defining blocks.

There are several ways to address the problem above, each with its advantages and limitations. One possibility is to attempt to syntactically detect all situations in which a statement allows a subsequent statement to execute, that is, all situations in which the former can only perform spawn steps. Like in Section 3.3.2 where we introduced special configurations of the form $C\surd$ (following Hennessy [36]) called "terminated configurations", we can introduce special "parallel configurations" $C\|$ stating that C can only spawn statements or discard terminated spawned statements, that is, C's statement can be executed in parallel with subsequent statements. Assuming such $C\|$ special configurations, we can remove the existing small-step SOS rule corresponding to rule (SmallStep-Spawn-Wait) in Section 3.5.4 and add instead the following rule:

$$\frac{\langle s_1, \sigma, \omega_{in}, \omega_{out}\rangle\| \qquad \langle s_2, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s'_2, \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle s_1\, \texttt{;}\, s_2, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s_1\, \texttt{;}\, s'_2, \sigma', \omega'_{in}, \omega'_{out}\rangle}$$

Alternatively, one can define $\|$ as a predicate only taking a statement instead of a configuration, and then move the $\|$ sequent out from the rule premise into a side condition. Nevertheless, the $\|$ configurations or predicates need to be defined such that they include only statements of the form $\texttt{spawn}\, s$ and $\texttt{spawn}\, s\, \texttt{;}\, \texttt{spawn}\, s'$ and $\texttt{let}\, x = i\, \texttt{in}\, \texttt{spawn}\, s$ and combinations of such statements. The idea is that such statements can safely be executed in parallel with whatever else follows them. For example, a statement of the form $\texttt{let}\, x = a\, \texttt{in}\, \texttt{spawn}\, s$ where a is not a value does not fall into this pattern. Exercise 97 asks the reader to further explore this approach.

One problem with the approach above is that it is quite non-modular. Indeed, the definition of $\|$ is strictly dependent upon the current language syntax. Adding or removing syntax to the language will require us to also revisit the definition of $\|$. Another and bigger problem with this approach is that it does not seem to work for other concurrent language constructs. For example, in many languages the creation of a thread is an expression (and not a statement) construct, returning to the calling context the new thread identifier as a value. The calling context can continue its execution using the returned value in parallel with the spawned thread. In our context,

for example, if `spawn` s returned an integer value, we would have to allow expressions of the form (`spawn` x `:=` x `+` 1) `<=` (`spawn` x `:=` x `+` 10) `+` x and be able to execute the two threads concurrently with the lookup for x and the evaluation of the `<=` boolean expression. It would be quite hard, if not impossible, to adapt the approach above to work with such common concurrent constructs which both return a value to the calling context and execute their code in parallel with the context.

Another way to address the loss of concurrent behaviors due to the syntactic constraints imposed by `let` on the `spawn` statements in its body, is to eliminate the `let` as soon as it is semantically unnecessary. Indeed, once the expression a is evaluated in a construct `let` x `=` a `in` s, the `let` is semantically unnecessary. The only reason we kept it was because it offered us an elegant syntactic means to keep track of the execution contexts both for its body statement and for the outside environment. Unfortunately, it is now precisely this "elegant syntactic means" that inhibits the intended concurrency of the `spawn` statement. One way to eliminate the semantically unnecessary `let` statements is to try to add a small-step SOS rule of the form:

$$\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s\ ;\ x := \sigma(x), \sigma[i/x], \omega_{in}, \omega_{out}\rangle$$

The idea here is to reduce the `let` statement to its body statement in a properly updated state, making sure that the state is recovered after the execution of the body statement. The infusion of the statement $x := \sigma(x)$ is a bit unorthodox, because $\sigma(x)$ can also be undefined; it works in our case here because we allow state updates of the form $\sigma[\bot/x]$, which have the effect to undefine σ in x (see Section 2.3.2), and the assignment rule generates such a state update. This trick is not crucial for our point here; if one does not like it, then one can split the rule above in two cases, one where $\sigma(x)$ is defined and one where is it undefined, and then add a special syntactic statement construct to undefine x in the second case. The real problem with this `let`-elimination approach is that it is simply *wrong* when we also have `spawn` statements in the language. For example, if s is a spawn statement then the `let` reduces to the `spawn` statement followed by the assignment to x; the small-step SOS rule for `spawn` allowing the spawned statement to be executed in parallel with its subsequent statement then kicks in and allows the assignment to x to be possibly evaluated before the spawned statement, thus resulting in a wrong behavior: the spawned statement should only see the x bound by its `let` construct, not the outside x (possibly undefined). A correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`'s body:

$$\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s[x'/x], \sigma[i/x'], \omega_{in}, \omega_{out}\rangle \qquad \text{where } x' \text{ is a fresh variable}$$

Besides having to provide and maintain (as the language changes) a substitution[3] operation and then having to pay linear or worse complexity each time a `let` is eliminated, and besides creating potentially unbounded garbage bindings in the state (e.g., the let statement can be inside a loop), the solution above appears to only solve our particular problem with our particular combination of `let` and `spawn`. It still does not seem to offer us a general solution for dealing with arbitrary constructs for concurrency, in particular with `spawn` constructs that evaluate to a value which is immediately returned to the calling context, as described a few paragraphs above. For example, while it allows for renaming the variable x into a fresh variable when the expression (`spawn` x `:=` x `+` 1) `<=` (`spawn` x `:=` x `+` 10) `+` x appears inside a `let` x `=` i `in` ... construct, we still have no clear way to evaluate both spawn expressions and the expression containing them concurrently.

The correct solution to deal with concurrency in small-step SOS is to place all the concurrent threads or processes in a syntactic "soup" where any particular thread or process, as well as any

[3] See Section 4.5.3.

communicating subgroups of them, can be picked and advanced one step. Since in our IMP++ language we want all threads to execute concurrently without any syntactic restrictions, we have to place all of them in some top-level "soup", making sure that each of them is unambiguously provided its correct execution environment. For example, if a thread was spawned from inside a let statement, then it should correctly see precisely the execution environment available at the place where it was spawned, possibly interacting with other threads seeing the same environment; it should not wrongly interfere with other threads happening to have had variables with the same names in their creation environments. This can be done either by using a substitution like above to guarantee that each bound variable is distinct, or by splitting the current state mapping variable identifiers to values into an *environment* mapping identifiers to memory locations and a *store* (or memory, or heap) mapping locations to values. In the latter case, each spawned thread would be packed together with its creation environment and all threads would share the store. The environment-store approach is the one that we will follow in Section 3.8 and in general in $\mathbb{K}$ in Chapter 5 and many other places in the book, so we will not insist on this approach in the context of small-step SOS. Regardless of whether one uses a substitution or an environment-store approach, to place all the threads in a top-level soup we need to devise a mechanism to propagate a newly spawned thread to the top level through each syntactic construct under which it can occur. This is tedious, but not impossible.

The morale of the discussion above is that putting features together in a language defined using small-step SOS is a highly non-trivial matter even when the language is trivial, like our IMP++. On the one hand, one has to non-modularly modify the configurations to hold all the required semantic data of all the features and then to modify all the rules to make sure that all these data is propagated and updated appropriately by each language construct. Addressing this problem is the main motivation of the MSOS approach discussed in Section 3.6. One the other hand, the desired feature interactions can require quite subtle changes to the semantics, which are sometimes hard to achieve purely syntactically. One cannot avoid the feeling that syntax is sometimes just too rigid, particularly when concurrency is concerned. This is actually one of the major motivations for the chemical abstract machine computational and semantic model discussed in Section 3.8.

Denotational Semantics

In order to accommodate all the semantic data needed by all the features, the denotation functions will now have the following types:

$$\begin{array}{l}[\![\_]\!] : AExp \to (State \times Buffer \rightharpoonup Int \cup \{error\} \times State \times Buffer) \\ [\![\_]\!] : BExp \to (State \times Buffer \rightharpoonup Bool \cup \{error\} \times State \times Buffer) \\ [\![\_]\!] : Stmt \to (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\})\end{array}$$

Moreover, since programs are now statements and since we want their denotation to only take an input buffer and to return a state, the remainder of the input buffer and the output buffer (recall that we deliberately discard the halting status), we replace the original denotation function of programs with the following (it is important to have a different name):

$$[\![\_]\!]_{\texttt{pgm}} : Pgm \to (Buffer \rightharpoonup State \times Buffer \times Buffer)$$

For example, the denotation of division becomes

$$[\![a_1 \,/\, a_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_1) \;/_{Int}\; 1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2), 3^{\text{rd}}(arg_2)) & \text{if} \quad 1^{\text{st}}(arg_1) \neq error \\ & \text{and} \quad 1^{\text{st}}(arg_2) \neq error \\ & \text{and} \quad 1^{\text{st}}(arg_2) \neq 0 \\ arg_1 & \text{if} \quad 1^{\text{st}}(arg_1) = error \\ (error, 2^{\text{nd}}(arg_2), 3^{\text{rd}}(arg_2)) & \text{if} \quad 1^{\text{st}}(arg_2) = error \\ & \text{or} \quad 1^{\text{st}}(arg_2) = 0 \end{cases}$$

where $arg_1 = [\![a_1]\!]\pi$ and $arg_2 = [\![a_2]\!](2^{\text{nd}}(arg_1), 3^{\text{rd}}(arg_1))$, the denotation of sequential composition becomes

$$[\![s_1 \;;\; s_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2),\; 3^{\text{rd}}(arg_1) : 3^{\text{rd}}(arg_2),\; 4^{\text{th}}(arg_2)) & \text{if} \quad 4^{\text{th}}(arg_1) = ok \\ arg_1 & \text{if} \quad 4^{\text{th}}(arg_1) = halting \end{cases}$$

where $arg_1 = [\![s_1]\!]\pi$ and $arg_2 = [\![s_2]\!](1^{\text{st}}(arg_1),\; 2^{\text{nd}}(arg_1))$, the denotational semantics of while loops $\texttt{while}\, b\, \texttt{do}\, s$ become the fixed-points of total functions

$$\begin{array}{rcl} \mathcal{F} & : & (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\}) \\ & \to & (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\}) \end{array}$$

defined as

$$\mathcal{F}(\alpha)(\pi) = \begin{cases} (1^{\text{st}}(arg_3),\; 2^{nd}(arg_3),\; 3^{\text{rd}}(arg_2) : 3^{\text{rd}}(arg_3),\; 4^{\text{th}}(arg_3)) & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{true} \\ & \text{and} \quad 4^{\text{th}}(arg_2) = ok \\ arg_2 & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{true} \\ & \text{and} \quad 4^{\text{th}}(arg_2) = halting \\ (2^{\text{nd}}(arg_1),\; 3^{\text{rd}}(arg_1),\; \epsilon,\; ok) & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{false} \\ (2^{\text{nd}}(arg_1),\; 3^{\text{rd}}(arg_1),\; \epsilon,\; halting) & \text{if} \quad 1^{\text{st}}(arg_1) = error \end{cases}$$

where $arg_1 = [\![b]\!]\pi$, $arg_2 = [\![s]\!](2^{\text{nd}}(arg_1),\; 3^{\text{rd}}(arg_1))$, and $arg_3 = \alpha(1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2))$, and the denotation of programs is defined as the function

$$[\![s]\!]_{\texttt{pgm}}\omega = (1^{\text{st}}(arg),\; 2^{\text{nd}}(arg),\; 3^{\text{rd}}(arg))$$

where $arg = [\![s]\!](\cdot, \omega)$, that is, the program statement is evaluated in the empty state and the halting flag is discarded.

Once all the changes above are applied in order to correctly handle the semantic requirements of the various features of IMP++, the denotational semantics of those features is relatively easy, basically a simple adaptation of their individual denotational semantics in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4, and 3.5.5. We let their precise definitions as an exercise to the reader (see Exercise 100). Note, however, that the resulting denotational semantics of IMP++ is still non-deterministic and non-concurrent. Because of this accepted limitation, we do not need to worry about the loss of concurrent behaviors due to the interaction between `spawn` and `let`.

3.5.7 Notes

The first to directly or indirectly pinpoint the limitations of plain SOS and denotational semantics when defining non-trivial languages were the inventors of alternative semantic frameworks, such as

Berry and Boudol [10, 11] who proposed the chemical abstract machine model (see Section 3.8), Felleisen and his collaborators [29, 99] who proposed evaluation contexts (see Section 3.7), and Mosses and his collaborators [60, 61, 62] who proposed the modular SOS approach (see Section 3.6). Among these, Mosses is perhaps the one who most vehemently criticized the lack of modularity of plain SOS, bringing as evidence natural features like the ones we proposed for IMP++, which require the structure of the configurations and implicitly the existing rules to change no matter whether there is any semantic interaction or not between the new and the old features.

The lack of modularity of SOS was visible even in Plotkin's original notes [70, 71], where he had to modify the definition of simple arithmetic expressions several times as his initial language evolved. Hennessy also makes it even more visible in his book [36]. Each time he adds a new feature, he also has to change the configurations and the entire existing semantics, similarly to each of our IMP extensions in this section. However, the lack of modularity of language definitional frameworks was not perceived as a major problem until late 1990es, partly because there were few attempts to give complete and rigorous semantics to real programming languages. Hennessy actually used each language extension as a pedagogical opportunity to teach the reader what new semantic components the feature needs and how and where those are located in each sequent. Note also that Hennessy's languages were rather simple and pure. His imperative language, called *WhileL*, was actually simpler even than our IMP (*WhileL* had no global variable declarations). Hennessy's approach was somewhat different from ours, namely he defined a series of different paradigmatic languages, each highlighting certain semantic aspects in a pure form, without including features that lead to complex semantic interactions (like the side effects, blocks with local variables, and threads as in our IMP++).

Wadler [97] proposes a language design experiment similar in spirit to our extensions of IMP in this section. His core language is purely functional, but some of its added features overlap with those of IMP and IMP++: state and side effects, output, non-determinism. Wadler's objective in [97] was to emphasize the elegance and usefulness of monads in implementing interpreters in pure functional languages like Haskell, but his motivations for doing so are similar to ours: the existing (implementation or semantic) approaches for programming language design are not modular enough. Monads can also be used to add modularity to denotational semantics, to avoid having to modify the mathematical domains into products of domains as we did in this section. The monadic approach to denotational semantics will be further discussed in Section 3.9. However, as also discussed in Section 3.4.3, the denotational approach to non-determinism and concurrency is to collect all possible behaviors, hereby programs evaluating to sets or lists of values. The same holds true in Wadler's monadic approach to implement interpreters in [97]. The problem with this is that the resulting interpreters or executable semantics are quite inefficient. Contrast that with the small-step SOS approach in Section 3.3 which allows us, for example using our implementation of it in Maude, to both execute programs non-deterministically, making a possibly non-deterministic choice at each step, and search for all possible behaviors of programs.

3.5.8 Exercises

Variable Increment

The exercises below refer to the IMP extension with variable increment discussed in Section 3.5.1.

Exercise 66. *Add variable increment to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the proof system at 1 above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement in Maude the rewriting logic theory at 2 above, like in Figure 3.9. To test it, add to the* IMP *programs in Figure 3.4 the following two:*

```
op sum++Pgm : -> Pgm .
eq sum++Pgm = (
     var m, n, s ;
     n := 100 ;
     while (++ m <= n) do (s := s + m)
) .
```

```
op nondet++Pgm : -> Pgm .
eq nondet++Pgm = (
     var x ;
     x := 1 ;
     x := ++ x / (++ x / x)
) .
```

The first program should have only one behavior, so, for example, both Maude commands below

```
rewrite < sum++Pgm > .
search  < sum++Pgm > =>! Cfg:Configuration .
```

should show the same result configuration, `< m |-> 101 & n |-> 100 & s |-> 5050 >`. *The second program should have (only) three different behaviors under big-step semantics; the first command below will show one of the three behaviors, but the second will show all three of them:*

```
rewrite < nondet++Pgm > .
search  < nondet++Pgm > =>! Cfg:Configuration .
```

The three behaviors captured by the big-step SOS discussed in Section 3.5.1 result in configurations `< x |-> 1 >`, `< x |-> 2 >`, *and* `< x |-> 3 >`. *As explained in Section 3.5.1, this big-step SOS should not be not able to expose the behaviors in which* `x` *is* `0` *and in which a division by zero takes place.*

Exercise 67. *Type* IMP *extended with variable increment:*

1. *Translate the big-step rule above into a rewriting logic rule that can be added to those in Figure 3.11 corresponding to the type system of* IMP*;*

2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the two additional programs in Example 66.*

Exercise 68. *Same as Exercise 66, but for small-step SOS instead of big-step SOS.*
☆ *Make sure that the small-step definition in Maude exhibits all five behaviors of program* `nondet++Pgm` *defined in Exercise 66 (the three behaviors exposed by the big-step definition in Maude in Exercise 66, plus one where* `x` *is* `0` *and one where the program gets stuck right before a division by zero).*

Exercise 69. *Same as Exercise 66, but for denotational semantics instead of big-step SOS.*
☆ *The definition in Maude should lack any non-determinism, so only one behavior should be observed for any program, including* `nondet++Pgm` *in Exercise 66.*

Input/Output

The exercises below refer to the IMP extension with input/output discussed in Section 3.5.2.

Exercise 70. *Add input/output to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, add to the* IMP *programs in Figure 3.4 the following three:*

```
op sumIOPgm : -> Pgm .
eq sumIOPgm = (
     var m, n, s ;
     n := read() ;
     while (m <= n) do (
       print(m) ;
       s := s + m ;
       m := m + 1
     ) ;
     print(s)
) .
```

```
op whileIOPgm : -> Pgm .
eq whileIOPgm = (
     var s ;
     s := 0 ;
     while not(read() <= 0) do
       s := s + read() ;
     print(s)
) .
```

```
op nondetIOStmt : -> Stmt .
eq nondetIOStmt = (
     print(read()
           / (read()
              / read()))
) .
```

The first two programs are deterministic, so both the rewrite and the search commands should only show one solution. The initial configuration in which the first program is executed should contain at least one integer in the input buffer, otherwise it does not evaluate; for example, the initial configuration `< sumIOPgm, 100 >` *yields a result configuration whose input buffer is empty and whose output buffer contains the numbers 0,1,2,...,100,5050. The initial configuration in which the second program is executed should eventually contain some 0 on an odd position in the input buffer; for example,* `< whileIOPgm,10:1:17:2:21:3:0:5:8:-2:-5:10 >` *yields a result configuration whose input buffer still contains the remaining input* `5:8:-2:-5:10` *and whose output buffer contains only the integer* `6`*. The third program, which is actually a statement, is non-deterministic. Unfortunately, big-step SOS misses behaviors because, as explained, it only achieves a non-deterministic choice semantics. For example, the commands*

```
rewrite < nondetIOStmt, .State, 10 : 20 : 30 > .
search  < nondetIOStmt, .State, 10 : 20 : 30 > =>! Cfg:Configuration .
```

yield configurations `< .State,epsilon,10 >` *and, respectively,* `< .State,epsilon,10 >` *and* `< .State,epsilon,15 >`*. The configuration whose output is 6 (i.e.,* $20/_{Int}(30/_{Int}10)$*) and the three undefined configurations due to division by zero are not detected.*

Exercise 71. *Type* IMP *extended with input/output:*

1. *Translate the discussed big-step SOS typing rules for input/output into corresponding rewriting logic rules that can be added to the already existing rewrite theory in Figure 3.11 corresponding to the type system of* IMP*;*

2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 70.*

Exercise 72. *Same as Exercise 70, but for small-step SOS instead of big-step SOS. Make sure that the resulting small-step SOS definition detects all six behaviors of* `nondetIOStmt` *when executed with the input buffer* `10:20:30`

Exercise 73. *Same as Exercise 70, but for denotational semantics instead of big-step SOS. Since our denotational semantics is not nondeterministic, only one behavior of* `nondetIOStmt` *is detected. Interestingly, since our denotational semantics of division was chosen to evaluate the two expressions*

in order, it turns out that the detected behavior is undefined (due to a division by zero). Note that although it also misses non-deterministic behaviors, big-step SOS can still detect (and even search for) valid behaviors of non-deterministic programs (see Exercise 70, where it generated a valid behavior by rewriting and found one additional behavior by searching).

Abrupt Termination

The exercises below refer to the IMP extension with abrupt termination discussed in Section 3.5.3.

Exercise 74. *Add abrupt termination to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.5.3. The resulting big-step SOS definition may be very slow when executed in Maude, even for small values of* `n` *(such as 2,3,4 instead of 100), which is normal (the search space is now much larger).*

Exercise 75. *Same as Exercise 74, but using a specific* `top` *construct as explained in Section 3.5.3 to catch the halting signal instead of the existing* `var` *which has a different purpose in the language.*

Exercise 76. *Type* IMP *extended with abrupt termination:*

1. *Translate the big-step SOS typing rule of* `halt` *into a corresponding rewriting logic rule that can be added to the already existing rewrite logic theory in Figure 3.11;*

2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 74.*

Exercise 77. *Same as Exercise 74, but for the small-step SOS instead of big-step SOS.*

Exercise 78. *Same as Exercise 74, but use the* `top` *construct approach instead of the rule* (SmallStep-Halting)*, to avoid wasting one computational step.*

Exercise 79. *One could argue that the introduction of the halting configurations* $\langle \texttt{halting}, \sigma \rangle$ *was unnecessary, because we could have instead used the already existing configurations of the form* $\langle \texttt{halt}, \sigma \rangle$*. Give an alternative small-step SOS definition of abrupt termination which does not add special halting configurations. Can we avoid the introduction of the* `top` *construct discussed above? Comment on the disadvantages of this approach.*

Exercise 80. *Same as Exercise 74, but for denotational semantics instead of big-step SOS.*

Exercise 81. *Same as Exercise 80, but modifying the denotation of assignment so that it is always undefined when the assigned variable has not been declared.*

Dynamic Threads

The exercises below refer to the IMP extension with dynamic threads discussed in Section 3.5.4.

Exercise 82. *Consider the two programs at the beginning of Section 3.5.4. Propose hypothetical executions of the second program corresponding to any of the seven possible values of* `x`*. What is the maximum value that* `s` *can have when the first program, as well as all its dynamically created threads, terminate? Is there some execution of the first program corresponding to each smaller value of* `s` *(but larger than or equal to 0)?*

Exercise 83. *Add dynamic threads to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.5.4. The resulting Maude big-step SOS definition may be slow on the first program for large initial values for n: even though it does not capture all possible behaviors, it still comprises many of them. For example, searching for all the result configurations when* $n = 10$ *gives 12 possible values for s, namely 55,56,...,66. On the other hand, the second program only shows one out of the seven behaviors, namely the one where x results in 111.*

Exercise 84. *Same as Exercise 83, but for the type system instead of big-step SOS.*

Exercise 85. *Same as Exercise 83, but for small-step SOS instead of big-step SOS. When representing the resulting small-step SOS into rewriting logic, the structural identity can be expressed as a rewriting logic equation, this way capturing faithfully the intended computational granularity of the small-step SOS.* ☆ *When implementing the resulting rewriting logic theory into Maude, this equation can either be added as a normal equation (using* `eq`*) or as an* `assoc` *attribute to the sequential composition construct. The former will only be applied from left-to-right when executed using Maude rewriting and search commands, but that is sufficient in our particular case here.*

Exercise 86. *Same as Exercise 83, but for denotational semantics instead of big-step SOS.*

Local Variables

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.5.

Exercise 87. ☆ *Implement in Maude the seven macros in the preamble of Section 3.5.5, which desugar blocks with local variable declarations into* `let` *constructs.*
Hint: In Maude, equations are applied in order. One should typically not rely on that, but in this case it may give us a simpler and more compact implementation.

Exercise 88. *Add blocks with local variables to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, modify the* IMP *programs in Figure 3.4 to use local variables and also add the two programs at the beginning of Section 3.5.5. To check whether the programs evaluate as expected, you can let some relevant variables purposely undeclared and bind them manually (to 0) in the initial state. When the programs terminate, you will see the new values of those variables. If you execute closed programs (i.e., programs declaring all the variables they use) then the resulting states will be empty, because our semantics of* `let` *recovers the state, so it will be difficult or impossible to know whether they executed correctly.*

Exercise 89. *Same as Exercise 88, but for the type system instead of big-step SOS.*

Exercise 90. *Same as Exercise 88, but for small-step SOS instead of big-step SOS.*

Exercise 91. *Same as Exercise 88, but for denotational semantics instead of big-step SOS.*

Putting Them All Together

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.6.

Exercise 92. *Define* IMP++ *using big-step SOS, assuming that abrupt termination applies to the entire program, no matter whether the abrupt termination signal has been issued from inside a spawned thread or from the main program, and assuming that nothing special is done to enhance the parallelism of* `spawn` *within* `let` *blocks:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, propose five tricky* IMP++ *programs. You programs will be added to our test-suite before grading this exercise. You get extra-points if your programs reveal limitations in the number of behaviors captured by other students' definitions.*

Exercise 93. *Same as Exercise 92, but assume that abrupt termination applies to the issuing thread only, letting the rest of the threads continue. The main program is considered to be a thread itself.*

Exercise 94. *Same as Exercise 92, but for the type system instead of big-step SOS.*

Exercise 95. *Same as Exercise 92, but for small-step SOS instead of big-step SOS.*

Exercise 96. *Same as Exercise 93, but for small-step SOS instead of big-step SOS.*

Exercise* 97. *Same as Exercise 95, but define and use "parallel configurations" of the form $C \| \ldots$ in order to enhance the parallelism of* `spawn` *statements from inside* `let` *blocks.*

Exercise* 98. *Same as Exercise 97, but eliminate the* `let` *construct when semantically unnecessary using a substitution operation (which needs to also be defined).*

Exercise* 99. *Same as Exercise 98, but use an environment-store approach to the state instead of substitution.*

Exercise 100. *Same as Exercise 92, but for denotational semantics instead of big-step SOS.*

3.6 Modular Structural Operational Semantics (MSOS)

Modular structural operational semantics (MSOS) was introduced, as its name implies, to address the non-modularity aspects of (big-step and/or small-step) SOS. As already seen in Section 3.5, there are several reasons why big-step and small-step SOS are non-modular, as well as several facets of non-modularity in general. In short, a definitional framework is *non-modular* when, in order to add a new feature to an existing language or calculus, one needs to revisit and change some or all of the already defined unrelated features. For example, recall the IMP extension with input/output in Section 3.5.2. We had to add new semantic components in the IMP configurations, both in the big-step and in the small-step SOS definitions, to hold the input/output buffers. That meant, in particular, that *all* the existing big-step and/or small-step SOS rules of IMP had to change. That was, at best, very inconvenient.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major endeavor, done once and for all, so having to go through the semantic rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions, in no particular order:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand side of a rule and a different or wrong language is defined.

- A modular semantic framework allows us to more easily reuse semantics of existing and probably already well-tested features in other languages or language extensions, thus increasing our productivity as language designers and our confidence in the correctness of the resulting semantic language definition.

- When designing a new language, as opposed to an existing well-understood language, one needs to experiment with features and combinations of features; having to do lots of unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

- There is a plethora of domain-specific languages these days, generated by the need to abstract away from low-level programming language details to important, domain-specific aspects of the application. Therefore, there is a need for rapid language design and experimentation for various domains. Moreover, domain-specific languages tend to be very dynamic, being added or removed features frequently as the domain knowledge evolves. It would be very nice to have the possibility to "drag-and-drop" language features in one's language, such as functions, exceptions, objects, etc.; however, in order for that to be possible, modularity is crucial.

Whether MSOS gives us such a desired and general framework for modular language design is and will probably always be open for debate. However, to our knowledge, MSOS is the first framework that explicitly recognizes the importance of modular language design and provides explicit support to achieve it in the context of SOS. Reduction semantics with evaluation contexts (see Section 3.7) was actually proposed before MSOS and also offers modularity in language semantic definitions, but

its modularity comes as a consequence of a different way to propagate reductions through language constructs and not as an explicit goal that it strives to achieve.

There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS. We actually generically call MSOS the small-step, implicitly-modular variant of MSOS (see Section 3.6.4). To bring modularity to SOS, MSOS proposes the following:

- Separate the syntax (i.e., the fragment of program under consideration) from the non-syntactic components in configurations, and treat them differently, as explained below;
- Make the transitions relate syntax to syntax only (as opposed to configurations), and hide the non-syntactic components in a transition label, as explained below;
- Encode in the transition label all the changes in the non-syntactic components of the configuration that need to be applied together with the syntactic reduction given by the transition;
- Use specialized notation in transition labels together with a discipline to refer to the various semantic components and to say that some of them stay unchanged; also, labels can be explicitly or implicitly shared by the conditions and the conclusion of a rule, elegantly capturing the idea that "changes are propagated" through desired language constructs.

A transition in MSOS is of the form

$$P \xrightarrow{\Delta} P'$$

where P and P' are programs or fragments of programs and Δ is a *label describing the semantic configuration components both before and after the transition.* Specifically, Δ is a record containing fields denoting the semantic components of the configuration. The preferred notation in MSOS for stating that in label Δ the semantic component associated to the field name field *before* the transition takes place is α is $\Delta = \{\mathsf{field} = \alpha, \ldots\}$. Similarly, the preferred notation for stating that the semantic component associated to field field *after* the transition takes place is β is $\Delta = \{\mathsf{field'} = \beta, \ldots\}$ (the field name is primed). For example, the second MSOS rule for variable assignment (when the assigned arithmetic expression is already evaluated) is (this is rule (MSOS-Asgn) in Figure 3.24):

$$x := i \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \texttt{skip} \quad \text{if } \sigma(x) \neq \bot$$

It is easy to desugar the rule above into a more familiar SOS rule of the form:

$$\langle x := i, \sigma \rangle \to \langle \texttt{skip}, \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot$$

The above is precisely the rule (SmallStep-Asgn) in the small-step SOS of IMP (see Figure 3.15). The MSOS rule is actually more modular than the SOS one, because of the "...", which says that everything else in the configuration stays unchanged. For example, if we want to extend the language with input/output language constructs as we did in Section 3.5.2, then new semantic components, namely the input and output buffers, need to be added to the configuration. Moreover, as seen in Section 3.5.2, the SOS rule above needs to be changed into a rule of the form

$$\langle x := i, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle \texttt{skip}, \sigma[i/x], \omega_{in}, \omega_{out} \rangle \quad \text{if } \sigma(x) \neq \bot$$

where ω_{in} and ω_{out} are the input and output buffers, respectively, which stay unchanged during the variable assignment operation, while the MSOS rule does not need to be touched.

To impose a better discipline on the use of labels, at the same time making the notation even more compact, MSOS splits the fields into three categories: *read-only*, *read-write*, and *write-only*. The field `state` above was read-write, meaning that the transition label can both read its value before the transition takes place and write its value after the transition takes place. Unlike the state, which needs to be both read and written, there are semantic configuration components that only need to be read, as well as ones that only need to be written. In these cases, it is recommended to use read-only or write-only fields.

Read-only fields are only inspected by the rule, but not modified, so they only appear unprimed in labels. For example, the following can be one of the MSOS rules for the `let` binding language construct in a pure functional language where expressions yield no side-effects:

$$\frac{e_2 \xrightarrow{\{\mathsf{env}=\rho[v_1/x],\ldots\}} e_2'}{\texttt{let}\, x = v_1\, \texttt{in}\, e_2 \xrightarrow{\{\mathsf{env}=\rho,\ldots\}} \texttt{let}\, x = v_1\, \texttt{in}\, e_2'}$$

Indeed, transitions do not modify the environment in a pure functional language. They only use it in a read-only fashion to lookup the variable values. A new environment is created in the premise of the rule above to reduce the body of the `let`, but neither of the transitions in the premise or in the conclusion of the rule change their environment. Note that this was not the case for IMP extended with `let` in Section 3.5.5, because there we wanted blocks and local variable declarations to desugar to `let` statements. Since we wanted blocks to be allowed to modify variables which were not declared locally, like it happens in conventional imperative and object-oriented languages, we needed an impure variant of `let`. As seen in Section 3.6.2, our MSOS definition of IMP++'s `let` uses a read-write attribute (the `state` attribute). We do not discuss read-only fields any further here.

Write-only fields are used to record data that is not analyzable during program execution, such as the output or the trace. Their names are always primed and they have a free monoid semantics—everything written on them is actually added to the end (see [62] for technical details). Consider, for example, the extension of IMP with an output (or print) statement in Section 3.5.2, whose MSOS second rule (after the argument is evaluated, namely rule (MSOS-Print) in Section 3.6.2) is:

$$\texttt{print(}i\texttt{)} \xrightarrow{\{\mathsf{output'}=i,\ \ldots\}} \texttt{skip}$$

Compare the rule above with the one below, which uses a read-write attribute instead:

$$\texttt{print(}i\texttt{)} \xrightarrow{\{\mathsf{output}=\omega_{out},\ \mathsf{output'}=\omega_{out}:i,\ \ldots\}} \texttt{skip}$$

Indeed, mentioning the ω_{out} like in the second rule above is unnecessary, error-prone (e.g., one may forget to add it to the primed field or may write $i : \omega_{out}$ instead of $\omega_{out} : i$), and non-modular (e.g., one may want to change the monoid construct, say to write $\omega_{out} \cdot i$ instead of $\omega_{out} : i$, etc.).

MSOS achieves modularity in two ways:

1. By making intensive use of the record comprehension notation "...", which, as discussed, indicates that more fields could follow but that they are not of interest. In particular, if the MSOS rule has no premises, like in the rules for the assignment and print statements discussed above, than the "..." says that the remaining contents of the label stays unchanged after the application of the transition; and

2. By reusing the same label or portion of label both in the premise and in the conclusion of an MSOS proof system rule. In particular, if "..." is used in the labels of both the premise and the conclusion of an MSOS rule, then all the occurrences of "..." stand for the same portion of label, that is, the same fields bound to the same semantic components.

For example, the following MSOS rules for first-statement reduction in sequential composition are equivalent and say that all the configuration changes generated by reducing s_1 to s_1' are propagated when reducing $s_1 \; ; \; s_2$ to $s_1' \; ; \; s_2$:

$$\frac{s_1 \xrightarrow{\Delta} s_1'}{s_1 \; ; \; s_2 \xrightarrow{\Delta} s_1' \; ; \; s_2}$$

$$\frac{s_1 \xrightarrow{\{...\}} s_1'}{s_1 \; ; \; s_2 \xrightarrow{\{...\}} s_1' \; ; \; s_2}$$

$$\frac{s_1 \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} s_1'}{s_1 \; ; \; s_2 \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} s_1' \; ; \; s_2}$$

Indeed, advancing the first statement in a sequential composition of statements one step has the same effect on the configuration as if the statement was advanced the same one step in isolation, without the other statement involved; said differently, the side effects are all properly propagated.

MSOS (the implicitly-modular variant of it, see Section 3.6.4) has been refined to actually allow for dropping such redundant labels like above from rules. In other words, if a label is missing from a transition then the *implicit label* is assumed: if the rule is unconditional then the implicit label is the identity label (in which the primed fields have the same values as the corresponding unprimed ones, etc.), but if the rule is conditional then the premise and the conclusion transitions share the same label, that is, they perform the same changes on the semantic components of the configuration. With this new notational convention, the most elegant and compact way to write the rule above in MSOS is:

$$\frac{s_1 \to s_1'}{s_1 \; ; \; s_2 \to s_1' \; ; \; s_2}$$

This is precisely the rule (MSOS-Seq-Arg1) in Figure 3.24, part of the MSOS semantics of IMP.

One of the important merits of MSOS is that it captures formally many of the tricks that language designers informally use to avoid writing awkward and heavy SOS definitions.

Additional notational shortcuts are welcome in MSOS if properly explained and made locally rigorous, without having to rely on other rules. For example, the author of MSOS finds the rule

$$x \xrightarrow{\{\mathsf{state},\ ...\}} \mathsf{state}(x) \quad \text{if } \mathsf{state}(x) \neq \bot$$

to be an acceptable variant of the lookup rule:

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot$$

despite the fact that, strictly speaking, $\mathsf{state}(x)$ does not make sense by itself (recall that state is a field name, not the state) and that field names are expected to be paired with their semantic components in labels. Nevertheless, there is only one way to make sense of this rule, namely to replace any use of state by its semantic contents, which therefore does not need to be mentioned.

A major goal when using MSOS to define languages or calculi is to write on the labels as little information as possible and to use the implicit conventions for the missing information. That is because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme case, if one uses only read/write fields in labels and mentions all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS with regards to modularity.

Recall the rules in Figure 3.16 for deriving the transitive closure $\rightarrow^\star$ of the small-step SOS relation $\rightarrow$. In order for two consecutive transitions to compose, the source configuration of the second had to be identical to the target configuration of the first. A similar property must also hold in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of category theory (see [62] for technical details on MSOS; see Section 2.10 for details on category theory), associating MSOS labels with morphisms in a special category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating, or desugaring MSOS definitions into SOS, as we implicitly implied when we discussed the MSOS rule for variable assignment above. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations including, besides the syntactic contents, the complete semantic contents extracted from the notational conventions in the label. The resulting SOS configurations will not have fields anymore, but will nevertheless contain all the semantic information encoded by them. For example, in the context of a language containing only a state and an output buffer as semantic components in its configuration (note that IMP++ contained an input buffer as well), the four rules discussed above for variable assignment, output, sequential composition, and lookup desugar, respectively, into the following conventional SOS rules:

$$\langle x := i, \sigma, \omega \rangle \rightarrow \langle \texttt{skip}, \sigma[i/x], \omega \rangle \quad \text{if } \sigma(x) \neq \bot$$

$$\langle \texttt{print(}i\texttt{)}, \sigma, \omega \rangle \rightarrow \langle \texttt{skip}, \sigma, \omega : i \rangle$$

$$\frac{\langle s_1, \sigma, \omega \rangle \rightarrow \langle s_1', \sigma', \omega' \rangle}{\langle s_1 \texttt{ ; } s_2, \sigma, \omega \rangle \rightarrow \langle s_1' \texttt{ ; } s_2, \sigma', \omega' \rangle}$$

$$\langle x, \sigma, \omega \rangle \rightarrow \langle \sigma(x), \sigma, \omega \rangle \quad \text{if } \sigma(x) \neq \bot$$

Recall that for unconditional MSOS rules the meaning of the missing label fields is "stay unchanged", while in the case of conditional rules the meaning of the missing fields is "same changes in conclusion as in the premise". In order for all the changes explicitly or implicitly specified by MSOS rules to apply, one also needs to provide an initial state for all the attributes, or in terms of SOS, an initial configuration. The initial configuration is often left unspecified in MSOS or SOS paper language definitions, but it needs to be explicitly given when one is concerned with executing the semantics. In our SOS definition of IMP in Section 3.3.2 (see Figure 3.15), we created the

appropriate initial configuration in which the top-level statement was executed using the proof system itself, more precisely the rule (SmallStep-Var) created a configuration holding a statement and a state from a configuration holding only the program. That is not possible in MSOS, because MSOS assumes that the structure of the label record does not change dynamically as the rules are applied. Instead, it assumes all the attributes given and fixed. Therefore, one has to explicitly state the initial values corresponding to each attribute in the initial state. However, in practice those initial values are understood and, consequently, we do not bother defining them. For example, if an attribute holds a list or a set, then its initial value is the empty list or set; if it holds a partial function, then its initial value is the partial function undefined everywhere; etc.

This way of regarding MSOS as a convenient front-end to SOS also supports the introduction of further notational conventions in MSOS if desired, like the one discussed above using $\mathsf{state}(x)$ instead of $\sigma(x)$ in the right-hand side of the transition, provided that one explains how such conventions are desugared when going from MSOS to SOS. Finally, the translation of MSOS into SOS also allows MSOS to borrow from SOS the reflexive/transitive closure $\rightarrow^\star$ of the one-step relation.

3.6.1 The MSOS of IMP

Figures 3.23 and 3.24 show the MSOS definition of IMP. There is not much to comment on the MSOS rules in these figures, except, perhaps, to note how compact and elegant they are compared to the corresponding SOS definition in Figures 3.14 and 3.15. Except for the three rules (MSOS-Lookup), (MSOS-Asgn), and (MSOS-Var), which make use of labels, they are as compact as they can be in any SOS-like setting for any language including the defined constructs. Also, the above-mentioned three rules only mention those components from the labels that they really need, so they allow for possible extensions of the language, like the IMP++ extension in Section 3.5.

The rule (MSOS-Var) is somehow different from the other rules that need the information in the label, in that it uses an attribute which has the type read-write but it only writes it without reading it. This is indeed possible in MSOS. The type of an attribute cannot be necessarily inferred from the way it is used in some of the rules, and not all rules must use the same attribute in the same way. One should explicitly clarify the type of each attribute before one gives the actual MSOS rules, and one is not allowed to change the attribute types dynamically, during derivations. Indeed, if the type of the `output` attribute in the MSOS rules for output above (and also in Section 3.6.2) were read-write, then the rules would wrongly imply that the output buffer will only store the last value, the previous ones being lost (this could be a desirable semantics in some cases, but not here).

Since the MSOS proof system in Figures 3.23 and 3.24 translates, following the informal procedure described above, in the SOS proof system in Figures 3.14 and 3.15, basically all the small-step SOS intuitions and discussions for IMP in Section 3.3.2 carry over here almost unchanged. In particular:

Definition 18. *We say that $C \rightarrow C'$ is derivable with the MSOS proof system in Figures 3.23 and 3.24, written* $\text{MSOS}(\text{IMP}) \vdash C \rightarrow C'$*, iff* $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow C'$ *(using the proof system in Figures 3.14 and 3.15). Similarly,* $\text{MSOS}(\text{IMP}) \vdash C \rightarrow^\star C'$ *iff* $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^\star C'$.

Note, however, that MSOS is more syntactic in nature than SOS, in that each of its reduction rules requires syntactic terms in both sides of the transition relation. In particular, that means that, unlike in SOS (see Exercise 52), in MSOS one does not have the option to dissolve statements from configurations anymore. Instead, one needs to reduce them to `skip` or some similar syntactic constant; if the original language did not have such a constant then one needs to invent one and add it to the original language or calculus syntax.

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Lookup)}$$

$$\frac{a_1 \to a_1'}{a_1 + a_2 \to a_1' + a_2} \qquad \text{(MSOS-Add-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 + a_2 \to a_1 + a_2'} \qquad \text{(MSOS-Add-Arg2)}$$

$$i_1 + i_2 \to i_1 +_{Int} i_2 \qquad \text{(MSOS-Add)}$$

$$\frac{a_1 \to a_1'}{a_1 \,/\, a_2 \to a_1' \,/\, a_2} \qquad \text{(MSOS-Div-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 \,/\, a_2 \to a_1 \,/\, a_2'} \qquad \text{(MSOS-Div-Arg2)}$$

$$i_1 \,/\, i_2 \to i_1 \,/_{Int}\, i_2 \quad \text{if } i_2 \neq 0 \qquad \text{(MSOS-Div)}$$

$$\frac{a_1 \to a_1'}{a_1 \texttt{<=} a_2 \to a_1' \texttt{<=} a_2} \qquad \text{(MSOS-Leq-Arg1)}$$

$$\frac{a_2 \to a_2'}{i_1 \texttt{<=} a_2 \to i_1 \texttt{<=} a_2'} \qquad \text{(MSOS-Leq-Arg2)}$$

$$i_1 \texttt{<=} i_2 \to i_1 \leq_{Int} i_2 \qquad \text{(MSOS-Leq)}$$

$$\frac{b \to b'}{\texttt{not}\, b \to \texttt{not}\, b'} \qquad \text{(MSOS-Not-Arg)}$$

$$\texttt{not true} \to \texttt{false} \qquad \text{(MSOS-Not-True)}$$

$$\texttt{not false} \to \texttt{true} \qquad \text{(MSOS-Not-False)}$$

$$\frac{b_1 \to b_1'}{b_1 \,\texttt{and}\, b_2 \to b_1' \,\texttt{and}\, b_2} \qquad \text{(MSOS-And-Arg1)}$$

$$\texttt{false and}\, b_2 \to \texttt{false} \qquad \text{(MSOS-And-False)}$$

$$\texttt{true and}\, b_2 \to b_2 \qquad \text{(MSOS-And-True)}$$

Figure 3.23: MSOS(IMP) — MSOS of IMP Expressions ($i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $a_1, a_1', a_2, a_2' \in \mathit{AExp}$; $b, b', b_1, b_1', b_2 \in \mathit{BExp}$; $\sigma \in \mathit{State}$).

$$\frac{a \to a'}{x := a \to x := a'} \qquad \text{(MSOS-Asgn-Arg2)}$$

$$x := i \xrightarrow{\{\text{state}=\sigma,\ \text{state'}=\sigma[i/x],\ ...\}} \texttt{skip} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Asgn)}$$

$$\frac{s_1 \to s_1'}{s_1 \,;\, s_2 \to s_1' \,;\, s_2} \qquad \text{(MSOS-Seq-Arg1)}$$

$$\texttt{skip} \,;\, s_2 \to s_2 \qquad \text{(MSOS-Seq-Skip)}$$

$$\frac{b \to b'}{\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \to \texttt{if}\, b' \,\texttt{then}\, s_1 \,\texttt{else}\, s_2} \qquad \text{(MSOS-If-Arg1)}$$

$$\texttt{if true then}\, s_1 \,\texttt{else}\, s_2 \to s_1 \qquad \text{(MSOS-If-True)}$$

$$\texttt{if false then}\, s_1 \,\texttt{else}\, s_2 \to s_2 \qquad \text{(MSOS-If-False)}$$

$$\texttt{while}\, b \,\texttt{do}\, s \to \texttt{if}\, b \,\texttt{then}\, (s \,;\, \texttt{while}\, b \,\texttt{do}\, s) \,\texttt{else skip} \qquad \text{(MSOS-While)}$$

$$\texttt{var}\, xl \,;\, s \xrightarrow{\{\text{state'}= xl \mapsto 0,\ ...\}} s \qquad \text{(MSOS-Var)}$$

Figure 3.24: MSOS(IMP) — MSOS of IMP Statements ($i \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List}\{\mathit{Id}\}$;$a, a' \in \mathit{AExp}$; $b, b' \in \mathit{BExp}$; $s, s_1, s_1', s_2 \in \mathit{Stmt}$; $\sigma \in \mathit{State}$).

3.6.2 The MSOS of IMP++

We next discuss the MSOS of IMP++, playing the same language design scenario as in Section 3.5: we first add each feature separately to IMP, as if that feature was the final extension of the language, and then we add all the features together and investigate the modularity of the resulting definition as well as possibly unexpected feature interactions.

Variable Increment

In MSOS, one can define the increment modularly:

$$\texttt{++}\, x \xrightarrow{\{\text{state}=\sigma,\ \text{state'}=\sigma[(\sigma(x)+_{\mathit{Int}}1)/x],\ ...\}} \sigma(x) +_{\mathit{Int}} 1 \qquad \text{(MSOS-Inc)}$$

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its premise(s).

Input/Output

MSOS can modularly support the input/output extension of IMP. We need to add new label attributes holding the input and the output buffers, say input and output, respectively, and then to add the corresponding rules for the input/output constructs. Note that the input attribute is read-write, while the output attribute is write-only. Here are the MSOS rules for input/output:

$$\texttt{read()} \xrightarrow{\{\text{input}=i:\omega,\ \text{input'}=\omega,\ ...\}} i \qquad \text{(MSOS-Read)}$$

$$\frac{a \to a'}{\texttt{print(}\,a\,\texttt{)} \to \texttt{print(}\,a'\,\texttt{)}} \qquad \text{(MSOS-Print-Arg)}$$

$$\texttt{print(}\,i\,\texttt{)} \xrightarrow{\{\mathsf{output'}=i,\ \ldots\}} \texttt{skip} \qquad \text{(MSOS-Print)}$$

Note that, since output is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If output was declared as a read-write attribute, then the label of the second rule above would have been $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$. A major implicit objective of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to the language. For example, if for some reason one declared output as a read-write attribute and then later on one decided to change the list construct for the output integer list from colon "$\_ : \_$" to something else, say "$\_ \cdot \_$", then one would need to change the label in the second rule above from $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$ to $\{\mathsf{output} = \omega, \mathsf{output'} = (\omega \cdot i), \ldots\}$. Therefore, in the spirit of enhanced modularity and clarity, the language designer using MSOS is strongly encouraged to use write-only (or read-only) attributes instead of read-write attributes whenever possible.

Notice the lack of expected duality between the rules (MSOS-Read) and (MSOS-Print) for input and for output above. Indeed, for all the reasons mentioned above, one would like to write the rule (MSOS-Read) more compactly and modularly as follows:

$$\texttt{read()} \xrightarrow{\{\mathsf{input}=i,\ \ldots\}} i$$

Unfortunately, this is not possible with the current set of label attributes provided by MSOS. However, there is no reason why MSOS could not be extended to include more attributes. For example, an attribute called "consumable" which would behave as the dual of write-only, i.e., it would only have an unprimed variant in the label holding a monoid (or maybe a group?) structure like the read-only attributes but it would consume from it whatever is matched by the rule label, would certainly be very useful in our case here. If such an attribute type were available, then our input attribute would be of that type and our MSOS rule for `read()` would be like the one above.

A technical question regarding the execution of the resulting MSOS definition is how to provide input to programs. Or, put differently, how to initialize configurations. One possibility is to assume that the user is fully responsible for providing the initial attribute values. This is, however, rather inconvenient, because the user would then always have to provide an empty state and an empty output buffer in addition to the desired input buffer in each configuration. A more convenient approach is to invent a special syntax allowing the user to provide precisely a program and an input to it, and then to automatically initialize all the attributes with their expected values. Let us pair a program p and an input ω for it using a configuration-like notation of the form $\langle p, \omega \rangle$. Then we can replace the rule (MSOS-Var) in Figure 3.24 with the following rule:

$$\langle\, \texttt{var}\ xl\ \texttt{;}\ s,\ \omega \,\rangle \xrightarrow{\{\mathsf{state'}= xl \mapsto 0,\ \mathsf{input'}=\omega,\ \ldots\}} s$$

Abrupt Termination

MSOS allows for a more modular semantics of abrupt termination than the more conventional semantic approaches discussed in Section 3.5.3. However, in order to achieve modularity, we need to

extend the syntax of IMP with a `top` construct, similarly to the small-step SOS variant discussed in Section 3.5.3. The key to modularity here is to use the labeling mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only field in the MSOS labels, called halting, which is *true* whenever the program needs to halt, otherwise it is *false*[4]. Then we can add the following two MSOS rules that set the halting field to *true*:

$$i_1 \texttt{ / } 0 \xrightarrow{\{\mathsf{halting}'=true,\ ...\}} i_1 \texttt{ / } 0 \qquad \text{(MSOS-Div-By-Zero)}$$

$$\texttt{halt} \xrightarrow{\{\mathsf{halting}'=true,\ ...\}} \texttt{halt} \qquad \text{(MSOS-Halt)}$$

As desired, it is indeed the case now that a fact of the form $s \xrightarrow{\{\mathsf{halting}'=true,\ ...\}} s'$ is derivable if and only if $s = s'$ and the next executable step in s is either a `halt` statement or a division-by-zero expression. If one does not like keeping the syntax unchanged when an abrupt termination takes place, then one can add a new syntactic construct, say `stuck` like in [62], and replace the right-hand-side configurations above with `stuck`; that does not conceptually change anything in what follows. The setting seems therefore perfect for adding a rule of the form

$$\frac{s \xrightarrow{\{\mathsf{halting}'=true,\ ...\}} s}{s \xrightarrow{\{\mathsf{halting}'=false,\ ...\}} \texttt{skip}}$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a `skip` statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-Seq) rule, and thus hide the actual halting status of the precondition. To properly capture the halting status, we define a top level statement construct like we discussed in the context of big-step and small-step SOS above, say `top` *Stmt*, modify the rule (MSOS-Var) from

$$\texttt{var}\ xl\ \texttt{;}\ s \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ ...\}} s$$

to

$$\texttt{var}\ xl\ \texttt{;}\ s \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ \mathsf{halting}=false,\ ...\}} \texttt{top}\ s$$

to mark the top level statement, and then finally include the following three rules:

$$\frac{s \xrightarrow{\{\mathsf{halting}'=false,\ ...\}} s'}{\texttt{top}\ s \xrightarrow{\{\mathsf{halting}'=false,\ ...\}} \texttt{top}\ s'} \qquad \text{(MSOS-Top-Normal)}$$

$$\texttt{top skip} \to \texttt{skip} \qquad \text{(MSOS-Top-Skip)}$$

$$\frac{s \xrightarrow{\{halting'=true,\ ...\}} s}{\texttt{top}\ s \xrightarrow{\{\mathsf{halting}'=false,\ ...\}} \texttt{skip}} \qquad \text{(MSOS-Top-Halting)}$$

[4]Strictly speaking, MSOS requires that the write-only attributes take values from a free monoid; if one wants to be faithful to that MSOS requirement, then one can replace *true* with some letter word and *false* with the empty word.

The use of a `top` construct like above seems unavoidable if we want to achieve modularity. Indeed, we managed to avoid it in the small-step SOS definition of abrupt termination in Section 3.5.3 (paying one relatively acceptable additional small-step to dissolve the halting configuration), because the halting configurations were explicitly, and thus non-modularly propagated through each of the language constructs, so the entire program reduced to a halting configuration whenever a division by zero or a `halt` statement was encountered. Unfortunately, that same approach does not work with MSOS (unless we want to break its modularity, like in SOS), because the syntax is not mutilated when an abrupt termination occurs. The halting signal is captured by the label of the transition. However, the label does not tell us when we are at the top level in order to dissolve the halting status. Adding a new label to hold the depth of the derivation, or at least whether we are the top or not, would require one to (non-modularly) change it in each rule. The use of an additional `top` construct like we did above appears to be the best trade-off between modularity and elegance here.

Note that, even though MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is *not* modular when applied in SOS via the translation. That is because adding a new attribute in the label means adding a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS technique above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Dynamic Threads

The small-step SOS rule for spawning threads discussed in Section 3.5.4 can be straightforwardly turned into MSOS rules:

$$\frac{s \to s'}{\texttt{spawn}\, s \to \texttt{spawn}\, s'} \qquad \text{(MSOS-Spawn-Arg)}$$

$$\texttt{spawn skip} \to \texttt{skip} \qquad \text{(MSOS-Spawn-Skip)}$$

$$\frac{s_2 \to s_2'}{\texttt{spawn}\, s_1 \texttt{ ; } s_2 \to \texttt{spawn}\, s_1 \texttt{ ; } s_2'} \qquad \text{(MSOS-Spawn-Wait)}$$

$$(s_1 \texttt{ ; } s_2) \texttt{ ; } s_3 \equiv s_1 \texttt{ ; } (s_2 \texttt{ ; } s_3) \qquad \text{(MSOS-Seq-Assoc)}$$

Even though the MSOS rules above are conceptually identical to the original small-step SOS rules, they are more modular because, unlike the former, they carry over unchanged when the configuration needs to change. Note that the structural identity stating the associativity of sequential composition, called (MSOS-Seq-Assoc) above, is still necessary.

Local Variables

Section 3.5.5 showed how blocks with local variables can be desugared into a uniform `let` construct, and also gave the small-step SOS rules defining the semantics of `let`. Those rules can be immediately adapted into the following MSOS rules:

$$\frac{a \to a'}{\texttt{let}\ x = a\ \texttt{in}\, s \to \texttt{let}\ x = a'\ \texttt{in}\, s} \qquad \text{(MSOS-Let-Exp)}$$

$$\frac{s \xrightarrow{\{\mathsf{state}=\sigma[i/x],\ \mathsf{state}'=\sigma',\ \ldots\}} s'}{\texttt{let}\ x = i\ \texttt{in}\ s \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state}'=\sigma'[\sigma(x)/x],\ \ldots\}} \texttt{let}\ x = \sigma'(x)\ \texttt{in}\ s'} \qquad \text{(MSOS-Let-Stmt)}$$

$$\texttt{let}\ x = i\ \texttt{in}\ \texttt{skip} \to \texttt{skip} \qquad \text{(MSOS-Let-Done)}$$

Like for the other features, the MSOS rules are more modular than their small-step SOS variants.

Since programs are now just ordinary (closed) expressions and they are now executed in the empty state, the rule (MSOS-Var), namely

$$\texttt{var}\ xl\ \texttt{;}\ s \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ \ldots\}} s$$

needs to change into a rule of the from

$$s \xrightarrow{\{\mathsf{state}'=\cdot,\ \ldots\}} ?$$

Unfortunately, regardless of what we place instead of "?", such a rule will not work. That is because there is nothing to prevent it to apply to any statement at any step during the reduction. To enforce it to happen only at the top of the program and only at the beginning of the reduction, we can wrap the original program (which is a statement) into a one-element configuration-like term $\langle s \rangle$. Then the rule (MSOS-Var) can be replaced with the following rule:

$$\langle s \rangle \xrightarrow{\{\mathsf{state}'=\cdot,\ \ldots\}} s$$

Putting Them All Together

The modularity of MSOS makes it quite easy to put all the features discussed above together and thus define the MSOS of IMP++. Effectively, we have to do the following:

1. We add the three label attributes used for the semantics of the individual features above, namely the read-write input attribute and the two write-only attributes output and halting.

2. We add all the MSOS rules of all the features above *unchanged* (nice!), except for the rule (MSOS-Var) for programs (which changed several times, anyway).

3. To initialize the label attributes, we add a pairing construct $\langle s, \omega \rangle$ like we did when we added the input/output extension of IMP, where s is a statement (programs are ordinary statements now) and ω is a buffer, and replace the rule (MSOS-Var) in Figure 3.24 with the following:

$$\langle s, \omega \rangle \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ \mathsf{input}'=\omega,\ \mathsf{halting}'=\mathit{false},\ \ldots\}} \texttt{top}\ s$$

It is important to note that the MSOS rules of the individual IMP extensions can be very elegantly combined into one language (IMP++). The rule (MSOS-Var) had to globally change in order to properly initialize the attribute values, but nothing had to be done in the MSOS rules of any of the features in order to put it together with the other MSOS rules of the other features.

Unfortunately, even though each individual feature has its intended semantics, the resulting IMP++ language does not. We still have the same semantic problems with regards to concurrency that we had in the context of small-step SOS in Section 3.5.6. More precisely, the concurrency

of `spawn` statements is limited to the blocks in which they appear. For example, a statement of the form (`let` x `=` i `in spawn` s_1) ; s_2 does not allow s_2 to be evaluated concurrently with s_1. The statement s_1 has to evaluate completely and then the `let` statement dissolved, before any step in s_2 can be performed. To fix this problem, we would have to adopt one of the solutions proposed in Section 3.5.6 in the context of small-step SOS. Thanks to its modularity, MSOS would make any of those solutions easier to implement than small-step SOS. Particularly, one can use the label mechanism to pass a spawned thread and its execution environment all the way to the top modularly, without having to propagate it explicitly through language constructs.

3.6.3 MSOS in Rewriting Logic

Like big-step and small-step SOS, we can also associate a conditional rewrite rule to each MSOS rule and hereby obtain a rewriting logic theory that faithfully (i.e., step-for-step) captures the MSOS definition. There could be different ways to do this. One way to do it is to first desugar the MSOS definition into a step-for-step equivalent small-step SOS definition as discussed above, and then use the faithful embedding of small-step SOS into rewriting logic discussed in Section 3.3.3. The problem with this approach is that the resulting small-step SOS definition, and implicitly the resulting rewriting logic definition, lack the modularity of the original MSOS definition. In other words, if one wanted to extend the MSOS definition with rules that would require global changes to its corresponding SOS definition (e.g., ones adding new semantic components into the label/configuration), then one would also need to manually incorporate all those global changes in the resulting rewriting logic definition.

We first show that any MSOS proof system, say MSOS, can be mechanically translated into a rewriting logic theory, say $\mathcal{R}_{\text{MSOS}}$, in such a way that two important aspects of the original MSOS definition are preserved: 1) the corresponding derivation relations are step-for-step equivalent, that is, MSOS $\vdash C \to C'$ if and only if $\mathcal{R}_{\text{MSOS}} \vdash \mathcal{R}_{C \to C'}$, where $\mathcal{R}_{C \to C'}$ is the corresponding syntactic translation of the MSOS sequent $C \to C'$ into a rewriting logic sequent; and 2) $\mathcal{R}_{\text{MSOS}}$ is as modular as MSOS. Second, we apply our generic translation technique to the MSOS formal system MSOS(IMP) defined in Section 3.6.1 and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to and as modular as MSOS(IMP). The modularity of MSOS(IMP) and of $\mathcal{R}_{\text{MSOS(IMP)}}$ will pay off when we extend IMP in Section 3.5. Finally, we show how $\mathcal{R}_{\text{MSOS(IMP)}}$ can be seamlessly defined in Maude, yielding another interpreter for IMP (in addition to those corresponding to the big-step and small-step SOS definitions of IMP in Sections 3.2.3 and 3.3.3).

Computationally and Modularly Faithful Embedding of MSOS into Rewriting Logic

Our embedding of MSOS into rewriting logic is very similar to that of small-step SOS, with one important exception: the non-syntactic components of the configuration are all grouped into a *record*, which is a multiset of *attributes*, each attribute being a pair associating appropriate semantic information to a *field*. This allows us to use multiset *matching* (or matching modulo associativity, commutativity, and identity) in the corresponding rewrite rules to extract the needed semantic information from the record, thus achieving not only a computationally equivalent embedding of MSOS into rewriting logic, but also one with the same degree of modularity as MSOS.

Formally, let us assume an arbitrary MSOS formal proof system. Let *Attribute* be a fresh sort and let *Record* be the sort **Bag{***Attribute***}** (that means that we assume all the infrastructure needed to define records as comma-separated bags, or multisets, of attributes). For each field

Field holding semantic contents *Contents* that appears unprimed or primed in any of the labels on any of the transitions in any of the rules of the MSOS proof system, let us assume an operation "*Field* = _ : *Contents* → *Attribute*" (the name of this postfix unary operation is "*Field* = _"). Finally, for each syntactic category *Syntax* used in any transition that appears anywhere in the MSOS proof system, let us define a configuration construct "$\langle \_, \_ \rangle$: *Syntax* × *Record* → *Configuration*". We are now ready to define our transformation of an MSOS rule into a rewriting logic rule:

1. Translate it into an SOS rule, as discussed right above Section 3.6.1; we could also go directly from MSOS rules to rewriting logic rules, but we would have to repeat most of the steps from MSOS to SOS that were already discussed;

2. Group all the semantic components in the resulting SOS configurations into a corresponding record, where each semantic component translates into a corresponding attribute using a corresponding label;

3. If the same attributes appear multiple times in a rule and their semantic components are not used anywhere in the MSOS rule, then replace them by a generic variable of sort *Record*;

4. Finally, use the technique in Section 3.3.3 to transform the resulting SOS-like rules into rewrite rules, tagging the left-hand-side configurations with the ∘ symbol.

Applying the steps above, the four MSOS rules discussed right above Section 3.6.1 (namely the ones for variable assignment, output, sequential composition of statements, and variable lookup) translate into the following rewriting logic rules:

$$\begin{aligned}
\circ\langle X := I, (\mathsf{state} = \sigma, \rho)\rangle &\to \langle\, \mathtt{skip}\,, (\mathsf{state} = \sigma[I/X], \rho)\rangle \\
\circ\langle\, \mathtt{output}\,(i), (\mathsf{output} = \omega,\ \rho)\rangle &\to \langle\, \mathtt{skip}\,, (\mathsf{output} = \omega i,\ \rho)\rangle \\
\circ\langle S_1\ ;\ S_2, \rho\rangle &\to \langle S_1'\ ;\ S_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle X, (\mathsf{state} = \sigma, \rho)\rangle &\to \langle \sigma(X), (\mathsf{state} = \sigma, \rho)\rangle
\end{aligned}$$

We use the same mechanism as for small-step SOS to obtain the reflexive and transitive many-step closure of the MSOS one-step transition relation. This mechanism was discussed in detail in Section 3.3.3; it essentially consists of adding a configuration marker $\star$ together with a rule "$\star\, Cfg \to \star\, Cfg'$ **if** $\circ\, Cfg \to Cfg'$" iteratively applying the one-step relation.

Theorem 5. ***(Faithful embedding of MSOS into rewriting logic)*** *For any MSOS definition* MSOS*, and any appropriate configurations C and C', the following equivalences hold:*

$$\text{MSOS} \vdash C \to C' \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\, \overline{C} \to^1 \overline{C'} \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\, \overline{C} \to \overline{C'}$$

$$\text{MSOS} \vdash C \to^\star C' \iff \mathcal{R}_{\text{MSOS}} \vdash \star\, \overline{C} \to \star\, \overline{C'}$$

where $\mathcal{R}_{\text{MSOS}}$ is the rewriting logic semantic definition obtained from MSOS *by translating each rule in* MSOS *as above. (Recall from Section 2.7 that $\to^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewriting logic. Also, as C and C' are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{C'}$ are ground terms.)*

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, let us elaborate on the apparent differences between MSOS and $\mathcal{R}_{\text{MSOS}}$ from a user perspective. The most visible difference is the SOS-like style of writing the rules, namely using configurations instead

sorts:
$\mathit{Attribute}$, Record, $\mathit{Configuration}$, $\mathit{ExtendedConfiguration}$
subsorts and aliases:
$\mathit{Record} = \mathbf{Bag}\{\mathit{Attribute}\}$
$\mathit{Configuration} < \mathit{ExtendedConfiguration}$
operations:
$\mathsf{state} = \_ \;:\; \mathit{State} \to \mathit{Attribute}$ // more fields can be added by need
$\langle \_, \_ \rangle \;:\; \mathit{AExp} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_, \_ \rangle \;:\; \mathit{BExp} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_, \_ \rangle \;:\; \mathit{Stmt} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_ \rangle \;:\; \mathit{Pgm} \to \mathit{Configuration}$
$\circ \_ \;:\; \mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce one step
$\star \_ \;:\; \mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce all steps
rule:
$\star\, \mathit{Cfg} \to \star\, \mathit{Cfg}'$ **if** $\circ\, \mathit{Cfg} \to \mathit{Cfg}'$ // where $\mathit{Cfg}, \mathit{Cfg}'$ are variables of sort $\mathit{Configuration}$

Figure 3.25: Configurations and infrastructure for the rewriting logic embedding of MSOS(IMP).

of labels, which also led to the inheritance of the $\circ$ mechanism from the embedding of SOS into rewriting logic. Therefore, the equivalent rewriting logic definition is slightly more verbose than the original MSOS definition. On the other hand, it has the advantage that it is more direct than the MSOS definition, in that it eliminates all the notational conventions. Indeed, if we strip MSOS out of its notational conventions and go straight to its essence, we find that that essence is precisely its use of multiset matching to modularly access the semantic components of the configuration. MSOS chose to do this on the labels, using specialized conventions for read-only/write-only/read-write components, while our rewriting logic embedding of MSOS does it in the configurations, uniformly for all semantic components. Where precisely this matching takes place is, in our view, less relevant. What is relevant and brings MSOS its modularity is that multiset matching *does* happen. Therefore, similarly to the big-step and small-step SOS representations in rewriting logic, we conclude that the rewrite theory $\mathcal{R}_{\mathrm{MSOS}}$ *is* MSOS, and *not* an encoding of it.

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, unfortunately, $\mathcal{R}_{\mathrm{MSOS}}$ (and implicitly MSOS) still lacks the main strengths of rewriting logic, namely context-insensitive and parallel application of rewrite rules. Indeed, the rules of $\mathcal{R}_{\mathrm{MSOS}}$ can only apply at the top, sequentially, so these rewrite theories corresponding to the faithful embedding of MSOS follow a rather poor rewriting logic style. Like for the previous embeddings, this is not surprising though and does not question the quality of our embeddings. All it says is that MSOS was not meant to have the capabilities of rewriting logic with regards to context-insensitivity and parallelism; indeed, all MSOS attempts to achieve is to address the lack of modularity of SOS and we believe that it succeeded in doing so. Unfortunately, SOS has several other major problems, which are discussed in Section 3.5.

$$\circ\langle X, (\mathsf{state} = \sigma, \rho)\rangle \to \langle \sigma(X), (\mathsf{state} = \sigma, \rho)\rangle \quad \textbf{if} \quad \sigma(X) \neq \bot$$

$$\begin{array}{rll}
\circ\langle A_1 + A_2, \rho\rangle \to \langle A_1' + A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 + A_2, \rho\rangle \to \langle A_1 + A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 + I_2, \rho\rangle \to \langle I_1 +_{Int} I_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle A_1 \,/\, A_2, \rho\rangle \to \langle A_1' \,/\, A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 \,/\, A_2, \rho\rangle \to \langle A_1 \,/\, A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \,/\, I_2, \rho\rangle \to \langle I_1 \,/_{Int}\, I_2, \rho\rangle & \textbf{if} & I_2 \neq 0
\end{array}$$

$$\begin{array}{rll}
\circ\langle A_1 \mathsf{<=} A_2, \rho\rangle \to \langle A_1' \mathsf{<=} A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle I_1 \mathsf{<=} A_2, \rho\rangle \to \langle I_1 \mathsf{<=} A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \mathsf{<=} I_2, \rho\rangle \to \langle I_1 \leq_{Int} I_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle\, \mathtt{not}\ B, \rho\rangle \to \langle\, \mathtt{not}\ B', \rho'\rangle & \textbf{if} & \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle\, \mathtt{not\ true}, \rho\rangle \to \langle\, \mathtt{false}, \rho\rangle & & \\
\circ\langle\, \mathtt{not\ false}, \rho\rangle \to \langle\, \mathtt{true}, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle B_1\ \mathtt{and}\ B_2, \rho\rangle \to \langle B_1'\ \mathtt{and}\ B_2, \rho'\rangle & \textbf{if} & \circ\langle B_1, \rho\rangle \to \langle B_1', \rho'\rangle \\
\circ\langle\, \mathtt{false\ and}\ B_2, \rho\rangle \to \langle\, \mathtt{false}, \rho\rangle & & \\
\circ\langle\, \mathtt{true\ and}\ B_2, \rho\rangle \to \langle B_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle X := A, \rho\rangle \to \langle X := A', \rho'\rangle & \textbf{if} & \circ\langle A, \rho\rangle \to \langle A', \rho'\rangle \\
\circ\langle X := I, (\mathsf{state} = \sigma, \rho)\rangle \to \langle\, \mathtt{skip}, (\mathsf{state} = \sigma[I/X], \rho)\rangle & \textbf{if} & \sigma(X) \neq \bot
\end{array}$$

$$\begin{array}{rll}
\circ\langle S_1\ ;\ S_2, \rho\rangle \to \langle S_1'\ ;\ S_2, \rho'\rangle & \textbf{if} & \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle\, \mathtt{skip}\ ;\ S_2, \rho\rangle \to \langle S_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle\, \mathtt{if}\ B\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho\rangle \to \langle\, \mathtt{if}\ B'\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho'\rangle & \textbf{if} & \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle\, \mathtt{if\ true\ then}\ S_1\ \mathtt{else}\ S_2, \rho\rangle \to \langle S_1, \rho\rangle & & \\
\circ\langle\, \mathtt{if\ false\ then}\ S_1\ \mathtt{else}\ S_2, \rho\rangle \to \langle S_2, \rho\rangle & &
\end{array}$$

$$\circ\langle\, \mathtt{while}\ B\ \mathtt{do}\ S, \rho\rangle \to \langle\, \mathtt{if}\ B\ \mathtt{then}\ (S\ ;\ \mathtt{while}\ B\ \mathtt{do}\ S)\ \mathtt{else\ skip}, \rho\rangle$$

$$\circ\langle\, \mathtt{var}\ Xl\ ;\ S\rangle \to \langle S, (\mathsf{state} = Xl \mapsto 0)\rangle$$

Figure 3.26: $\mathcal{R}_{\mathrm{MSOS(IMP)}}$: the complete MSOS of IMP in rewriting logic.

MSOS of IMP in Rewriting Logic

We here discuss the complete MSOS definition of IMP in rewriting logic, obtained by applying the faithful embedding technique discussed above to the MSOS definition of IMP in Section 3.6.1. Figure 3.25 gives an algebraic definition of configurations as well as needed additional record infrastructure; all the sorts, operations, and rules in Figure 3.25 were already discussed either above or in Section 3.3.3. Figure 3.26 gives the rules of the rewriting logic theory $\mathcal{R}_{\text{MSOS(IMP)}}$ that is obtained by applying the procedure above to the MSOS of IMP in Figures 3.23 and 3.24. Like before, we used the rewriting logic convention that variables start with upper-case letters; if they are greek letters, then we use a similar but larger symbol (e.g., σ instead of σ for variables of sort *State*, or ρ instead of ρ for variables of sort *Record*). The following corollary of Theorem 5 establishes the faithfulness of the representation of the MSOS of IMP in rewriting logic:

Corollary 4. $\text{MSOS(IMP)} \vdash C \to C' \iff \mathcal{R}_{\text{MSOS(IMP)}} \vdash \circ\overline{C} \to \overline{C'}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system MSOS(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{MSOS(IMP)}}$; the two are faithfully equivalent. Moreover, by the discussion following Theorem 5, $\mathcal{R}_{\text{MSOS(IMP)}}$ is also as modular as MSOS(IMP). This will be further emphasized in Section 3.5, where we will extend IMP with several features, some of which requiring more attributes.

☆ Maude Definition of IMP MSOS

Figure 3.27 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{MSOS(IMP)}}$ in Figures 3.26 and 3.25. The Maude module `IMP-SEMANTICS-MSOS` in Figure 3.27 is executable, so Maude, through its rewriting capabilities, yields an MSOS interpreter for IMP the same way it yielded big-step and small-step SOS interpreters in Sections 3.2.3 and 3.3.3, respectively; for example, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 7132 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, state = (n |-> 0 , s |-> 5050) >
```

Note that the rewrite command above took the same number of rewrite steps as the similar command executed on the small-step SOS of IMP in Maude discussed in Section 3.3.3, namely 7132. This is not unexpected, because matching is not counted as rewrite steps, no matter how complex it is.

Like for the big-step and small-step SOS definitions in Maude, one can also use any of the general-purpose tools provided by Maude on the MSOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. The same number of states as in the case of small-step SOS will be generated by this search command, namely 1509.

```
mod IMP-CONFIGURATIONS-MSOS is including IMP-SYNTAX + STATE .
  sorts Attribute Record Configuration ExtendedConfiguration .
  subsort Attribute < Record .
  subsort Configuration < ExtendedConfiguration .
  op empty : -> Record .
  op _,_ : Record Record -> Record [assoc comm id: empty] .
  op state`=_ : State -> Attribute .
  op <_,_> : AExp Record -> Configuration .
  op <_,_> : BExp Record -> Configuration .
  op <_,_> : Stmt Record -> Configuration .
  op <_> : Pgm -> Configuration .
  op o_ : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  op *_ : Configuration -> ExtendedConfiguration [prec 80] .   --- all steps
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-MSOS is including IMP-CONFIGURATIONS-MSOS .
  var X : Id .   var R R' : Record . var Sigma Sigma' : State .
  var I I1 I2 : Int .  var Xl : List{Id} .  var S S1 S1' S2 : Stmt .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 B2' : BExp .

 crl o < X,(state = Sigma, R) > => < Sigma(X),(state = Sigma, R) >
  if Sigma(X) =/=Bool undefined .
 crl o < A1 + A2,R > => < A1' + A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 + A2,R > => < A1 + A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 + I2,R > => < I1 +Int I2,R > .
 crl o < A1 / A2,R > => < A1' / A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 / A2,R > => < A1 / A2',R' > if o < A2,R > => < A2',R' > .
 crl o < I1 / I2,R > => < I1 /Int I2,R > if I2 =/=Bool 0 .
 crl o < A1 <= A2,R > => < A1' <= A2,R' > if o < A1,R > => < A1',R' > .
 crl o < I1 <= A2,R > => < I1 <= A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 <= I2,R > => < I1 <=Int I2,R > .
 crl o < not B,R > => < not B',R' > if o < B,R > => < B',R' > .
  rl o < not  true,R > => < false,R > .
  rl o < not false,R > => <  true,R > .
 crl o < B1 and B2,R > => < B1' and B2,R' > if o < B1,R > => < B1',R' > .
  rl o < false and B2,R > => < false,R > .
  rl o <  true and B2,R > => < B2,R > .
 crl o < X := A,R > => < X := A',R' > if o < A,R > => < A',R' > .
 crl o < X := I,(state = Sigma, R) > => < skip,(state = Sigma[I / X], R) >
  if Sigma(X) =/=Bool undefined .
 crl o < S1 ; S2,R > => < S1' ; S2,R' > if o < S1,R > => < S1',R' > .
  rl o < skip ; S2,R > => < S2,R > .
 crl o < if B then S1 else S2,R > => < if B' then S1 else S2,R' > if o < B,R > => < B',R'  > .
  rl o < if  true then S1 else S2,R > => < S1,R > .
  rl o < if false then S1 else S2,R > => < S2,R > .
  rl o < while B do S,R > => < if B then (S ; while B do S) else skip,R > .
  rl o < var Xl ; S > => < S,(state = Xl |-> 0) > .
endm
```

Figure 3.27: The MSOS of IMP in Maude, including the definition of configurations.

3.6.4 Notes

Modular Structural Operational Semantics (MSOS) was introduced in 1999 by Mosses [60] and since then mainly developed by himself and his collaborators (e.g., [61, 62, 63]). In this section we used the implicitly-modular variant of MSOS introduced in [63], which, as acknowledged by the authors of [63], was partly inspired from discussions with us[5]. To be more precise, we used a slightly simplified version of implicitly-modular MSOS here. In MSOS in its full generality, one can also declare some transitions *unobservable*; to keep the presentation simpler, we here omitted all the observability aspects of MSOS.

The idea of our representation of MSOS into rewriting logic adopted in this section is taken over from Serbanuta *et al.* [85]. At our knowledge, Meseguer and Braga [50] give the first representation of MSOS into rewriting logic. The representation in [50] also led to the development of the Maude MSOS tool [18], which was the core of Braga's doctoral thesis. What is different in the representation of Meseguer and Braga in [50] from ours is that the former uses two different types of configuration wrappers, one for the left-hand side of the transitions and one for the right-hand side; this was already discussed in Section 3.3.4.

3.6.5 Exercises

Exercise 101. *Redo all the exercises in Section 3.3.5 but for the MSOS of* IMP *discussed in Section 3.6.1 instead of its small-step SOS in Section 3.3.2. Skip Exercises 52, 53 and 60, since the SOS proof system there drops the syntactic components in the RHS configurations in transitions, making it unsuitable for MSOS. For the MSOS variant of Exercise 54, just follow the same non-modular approach as in the case of SOS and not the modular MSOS approach discussed in Section 3.6.2 (Exercise 104 addresses the modular MSOS variant of abrupt termination).*

Exercise 102. *Same as Exercise 68, but for MSOS instead of small-step SOS: add variable increment to* IMP*, like in Section 3.6.2.*

Exercise 103. *Same as Exercise 72, but for MSOS instead of small-step SOS: add input/output to* IMP*, like in Section 3.6.2.*

Exercise 104. *Same as Exercise 77, but for MSOS instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.6.2.*

Exercise 105. *Same as Exercise 85, but for MSOS instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.6.2.*

Exercise 106. *Same as Exercise 90, but for MSOS instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.6.2.*

Exercise 107. *This exercise asks to define* IMP++ *in MSOS, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the MSOS of* IMP++ *discussed in Section 3.6.2 instead of its small-step SOS in Section 3.5.6.*

[5] In fact, drafts of this book that preceded [63] dropped the implicit labels in MSOS rules for notational simplicity; Mosses found that simple idea worthwhile formalizing within MSOS.

3.7 Reduction Semantics with Evaluation Contexts

The small-step SOS/MSOS approaches discussed in Sections 3.3 and 3.6 define a language semantics as a proof system whose rules are mostly conditional. The conditions of such rules allow to implicitly capture the program execution context as a *proof context*. This shift of focus from the informal notion of execution context to the formal notion of proof context has a series of advantages and it was, to a large extent, the actual point of formalizing language semantics using SOS. However, as the complexity of programming languages increased, in particular with the adoption of control-intensive statements like call/cc (see Chapter 12) that can arbitrarily change the execution context, the need for an explicit representation of the execution context as a first-class citizen in the language semantics also increased. Reduction semantics with evaluation contexts (RSEC) is a variant of small-step SOS where the evaluation context may appear explicit in the term being reduced.

In an RSEC language definition one starts by defining the syntax of *evaluation contexts*, or simply just *contexts*, which is typically done by means of a context-free grammar (CFG). A context is a program or a fragment of program with a *hole*, where the hole, which is written $\Box$, is a placeholder for where the next computational step can take place. If c is an evaluation context and e is some well-formed appropriate fragment (expression, statement, etc.), then $c[e]$ is the program or fragment obtained by replacing the hole of c by e. Reduction semantics with evaluation contexts relies on a tacitly assumed (but rather advanced) parsing mechanism that takes a program or a fragment p and decomposes it into a context c and a subprogram or fragment e, called a *redex*, such that $p = c[e]$. This decomposition process is called *splitting* (of p into c and e). The inverse process, composing a redex e and a context c into a program or fragment p, is called *plugging* (of e into c). These splitting/plugging operations are depicted in Figure 3.28.

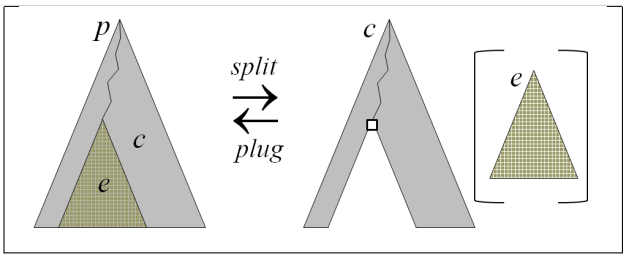

Figure 3.28: Decomposition of syntactic term p into context c and redex e, written $p = c[e]$: we say p *splits* into c and e, or e *plugs* into c yielding p. These operations are assumed whenever needed.

Consider a language with arithmetic/Boolean expressions and statements like our IMP language in Section 3.1. A possible CFG definition of evaluation contexts for such a language may include the following productions (the complete definition of IMP evaluation contexts is given in Figure 3.30):

$$
\begin{array}{rcl}
\mathit{Context} & ::= & \Box \\
& | & \mathit{Context} \texttt{ <= } \mathit{AExp} \\
& | & \mathit{Int} \texttt{ <= } \mathit{Context} \\
& | & \mathit{Id} \texttt{ := } \mathit{Context} \\
& | & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\
& | & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \\
& | & \ldots
\end{array}
$$

Note how the intended evaluation strategies of the various language constructs are reflected in the definition of evaluation contexts: `<=` is sequentially strict ($\Box$ allowed to go into the first subexpression until evaluated to an *Int*, then into the second subexpression), `:=` is strict only in its second argument while the sequential composition and the conditional are strict only in their first arguments. If one thinks of language constructs as operations taking a certain number

of arguments of certain types, then note that the operations appearing in the grammar defining evaluation contexts are *different* from their corresponding operations appearing in the language syntax; for example, Id `:=` *Context* is different from Id `:=` *AExp* because the former takes a context as second argument while the latter takes an arithmetic expression.

Here are some examples of correct evaluation contexts for the grammar above:

$\Box$

3 `<=` $\Box$

$\Box$ `<=` 3

$\Box$ `;` x `:=`5, where x is any variable

`if` $\Box$ `then` s_1 `else` s_2, where s_1 and s_2 are any well-formed statements

Here are some examples of incorrect evaluation contexts:

$\Box$ `<=` $\Box$ — a context can have only one hole

x `<=` 3 — a context must contain a hole

x `<=` $\Box$ — the first argument of `<=` must be an integer number in order to allow the hole in the second argument

x `:=`5 `;` $\Box$ — the hole can only appear in the first statement in a sequential composition

$\Box$ `:=`5 — the hole cannot appear as first argument of `:=`

`if` x `<=`7 `then` $\Box$ `else` x `:=`5 — the hole is only allowed in the condition of a conditional

Here are some examples of decompositions of syntactic terms into a context and a redex (recall that in this book we can freely use parentheses for disambiguation; here we enclose evaluation contexts in parentheses for clarity):

$7 = (\Box)[7]$

$3 \texttt{<=} x = (3 \texttt{<=} \Box)[x] = (\Box \texttt{<=} x)[3] = (\Box)[3 \texttt{<=} x]$

$3 \texttt{<=} (2 \texttt{+} x) \texttt{+} 7 = (3 \texttt{<=} \Box \texttt{+} 7)[2 \texttt{+} x] = (\Box \texttt{<=} (2 \texttt{+} x) \texttt{+} 7)[3] = \ldots$

For simplicity, we consider only one type of context in this section, but in general one can have various types, depending upon the types of their holes and of their result.

Reduction semantics with evaluation contexts tends to be a purely syntactic definitional framework (following the slogan "everything is syntax"). If semantic components are necessary in a particular definition, then they are typically "swallowed by the syntax". For example, if one needs a state as part of the configuration for a particular language definition (like we need for our hypothetical IMP language discussed here), then one adds a context production of the form

$$\mathit{Context} \quad ::= \quad \langle \mathit{Context}, \mathit{State} \rangle$$

where the *State*, an inherently semantic entity, becomes part of the evaluation context. Note that once one adds additional syntax to evaluation contexts that does not correspond to constructs in the syntax of the original language, such as our pairing of a context and a state above, one needs to also extend the original syntax with corresponding constructs, so that the parsing-like mechanism decomposing a syntactic term into a context and a redex can be applied. In our case, the

production above suggests that a pairing configuration construct of the form $\langle \mathit{Stmt}, \mathit{State} \rangle$, like for SOS, also needs to be defined. Unlike in SOS, we do not need configurations pairing other syntactic categories with a state, such as $\langle \mathit{AExp}, \mathit{State} \rangle$ and $\langle \mathit{BExp}, \mathit{State} \rangle$; the reason is that, unlike in SOS, transitions with left-hand-side configurations $\langle \mathit{Stmt}, \mathit{State} \rangle$ are not derived anymore from transitions with left-hand-side configurations of the form $\langle \mathit{AExp}, \mathit{State} \rangle$ or $\langle \mathit{BExp}, \mathit{State} \rangle$.

Evaluation contexts are defined in such a way that whenever e is reducible, $c[e]$ is also reducible. For example, consider the term $c[i_1 \texttt{<=} i_2]$ stating that expression $i_1 \texttt{<=} i_2$ is in a proper evaluation context. Since $i_1 \texttt{<=} i_2$ reduces to $i_1 \leq_{\mathit{Int}} i_2$, we can conclude that $c[i_1 \texttt{<=} i_2]$ reduces to $c[i_1 \leq_{\mathit{Int}} i_2]$. Therefore, in reduction semantics with evaluation contexts we can define the semantics of `<=` using the following rule:

$$c[i_1 \texttt{<=} i_2] \to c[i_1 \leq_{\mathit{Int}} i_2]$$

This rule is actually a rule schema, containing one rule instance for each concrete integers i_1, i_2 and for each appropriate evaluation context c. For example, here are instances of this rule when the context is $\Box$, $\texttt{if } \Box \texttt{ then skip else } x \texttt{:=} 5$, and $\langle \Box, (x \mapsto 1, y \mapsto 2) \rangle$, respectively:

$$\begin{gathered} i_1 \texttt{<=} i_2 \to i_1 \leq_{\mathit{Int}} i_2 \\ \texttt{if } (i_1 \texttt{<=} i_2) \texttt{ then skip else } x \texttt{:=} 5 \to \texttt{if } (i_1 \leq_{\mathit{Int}} i_2) \texttt{ then skip else } x \texttt{:=} 5 \\ \langle i_1 \texttt{<=} i_2 \ , \ (x \mapsto 1, y \mapsto 2) \rangle \to \langle i_1 \leq_{\mathit{Int}} i_2 \ , \ (x \mapsto 1, y \mapsto 2) \rangle \end{gathered}$$

What is important to note here is that propagation rules, such as (MSOS-Leq-Arg1) and (MSOS-Leq-Arg2) in Figure 3.23, are not necessary anymore when using evaluation contexts, because the evaluation contexts already achieve the role of the propagation rules.

To reflect the fact that reductions take place only in appropriate contexts, RSEC typically introduces a rule schema of the form:

$$\frac{e \to e'}{c[e] \to c[e']} \qquad \text{(RSEC-Characteristic-Rule)}$$

where e, e' are well-formed fragments and c is any appropriate evaluation context (i.e., such that $c[e]$ and $c[e']$ are well-formed programs or fragments of program). This rule is called the *characteristic rule* of RSEC. When this rule is applied, we say that e *reduces to* e' *in context* c. If one picks c to be the empty context $\Box$, then $c[e]$ is e and thus the characteristic rule is useless; for that reason, the characteristic rule may be encountered with a side condition "if $c \neq \Box$". Choosing good strategies to search for splits of terms into contextual representations can be a key factor in obtaining efficient implementations of RSEC execution engines.

The introduction of the characteristic rule allows us to define reduction semantics of languages or calculi quite compactly. For example, here are all the rules needed to completely define the semantics of the comparison (`<=`), sequential composition (`;`) and conditional (`if`) language constructs for which we defined evaluation contexts above:

$$\begin{aligned} i_1 \texttt{<=} i_2 &\to i_1 \leq_{\mathit{Int}} i_2 \\ \texttt{skip ; } s_2 &\to s_2 \\ \texttt{if true then } s_1 \texttt{ else } s_2 &\to s_1 \\ \texttt{if false then } s_1 \texttt{ else } s_2 &\to s_2 \end{aligned}$$

The characteristic rule tends to be the only conditional rule in an RSEC, in the sense that the remaining rules take no reduction premises (though they may still have side conditions). Moreover,

as already pointed out, the characteristic rule is actually unnecessary, because one can very well replace each rule $l \to r$ by a rule $c[l] \to c[r]$. The essence of reduction semantics with evaluation contexts is not its characteristic reduction rule, but its specific approach to defining evaluation contexts as a grammar and then using them as an explicit part of languages or calculi definitions. The characteristic reduction rule can therefore be regarded as "syntactic sugar", or convenience to the designer allowing her to write more compact definitions.

To give the semantics of certain language constructs, one may need to access specific information that is stored inside an evaluation context. For example, consider a term $\langle x \texttt{<=} 3\ ,\ (x \mapsto 1, y \mapsto 2)\rangle$, which can be split as $c[x]$, where c is the context $\langle \Box \texttt{<=} 3\ ,\ (x \mapsto 1, y \mapsto 2)\rangle$. In order to reduce $c[x]$ to $c[1]$ as desired, we need to look inside c and find out that the value of x in the state held by c is 1. Therefore, following the purely syntactic style adopted so far in this section, the reduction semantics with evaluation contexts rule for variable lookup in our case here is the following:

$$\langle c, \sigma\rangle[x] \to \langle c, \sigma\rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot$$

Indeed, the same way we add as much structure as needed in ordinary terms, we can add as much structure as needed in evaluation contexts. Similarly, below is the rule for variable assignment:

$$\langle c, \sigma\rangle[x \texttt{ := } i] \to \langle c, \sigma[i/x]\rangle[\texttt{ skip }] \quad \text{if } \sigma(x) \neq \bot$$

Note that in this case both the context and the redex were changed by the rule. In fact, as discussed in Section 3.10, one of the major benefits of reduction semantics with evaluation contexts consists in precisely the fact that one can arbitrarily modify the evaluation context in rules; this is crucial for giving semantics to control-intensive language constructs such as call/cc.

Splitting of a term into an evaluation context and a redex does not necessarily need to take place at the top of the left-hand side of a rule. For example, the following is an alternative way to give reduction semantics with evaluation contexts to variable lookup and assignment:

$$\begin{array}{c} \langle c[x], \sigma\rangle \to \langle c[\sigma(x)], \sigma\rangle \quad \text{if } \sigma(x) \neq \bot \\ \langle c[x \texttt{ := } i], \sigma\rangle \to \langle c[\texttt{ skip }], \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot \end{array}$$

Note that, even if one decides to follow this alternative style, one still needs to include the production $\mathit{Context} ::= \langle \mathit{Context}, \mathit{State}\rangle$ to the evaluation context CFG if one wants to write rules as $c[i_1 \texttt{<=} i_2] \to c[i_1 \leq_{\mathit{Int}} i_2]$ or to further take advantage of the characteristic rule and write elegant and compact rules such as $i_1 \texttt{<=} i_2 \to i_1 \leq_{\mathit{Int}} i_2$. If one wants to completely drop evaluation context productions that mix syntactic and semantic components, such as $\mathit{Context} ::= \langle \mathit{Context}, \mathit{State}\rangle$, then one may adopt one of the styles discussed in Exercises 108 and 109, respectively, though one should be aware of the fact that those styles also have their disadvantages.

Figure 3.29 shows a reduction sequence using the evaluation contexts and the rules discussed so far. In Figure 3.29, we used the following (rather standard) notation for instantiated contexts whenever we applied the characteristic rule: the redex is placed in a box replacing the hole of the context. For example, the fact that expression $3 \texttt{<=} x$ is split into contextual representation $(3 \texttt{<=} \Box)[x]$ is written compactly and intuitively as $3 \texttt{<=} \boxed{x}$. Note that the evaluation context changes almost at each step during the reduction sequence in Figure 3.29.

Like in small-step SOS and MSOS, we can also transitively and reflexively close the one-step transition relation $\to$. As usual, we let $\to^\star$ denote the resulting multi-step transition relation.

$$
\begin{array}{ll}
 & \langle \boxed{x := 1}\ ;\ y := 2\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 0, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{\texttt{skip}\ ;\ y := 2}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{y := 2}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \texttt{skip}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ \boxed{x}\ <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ 1 <= \boxed{y}\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ \boxed{1 <= 2}\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if true then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle x := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{skip}\ ,\ (x \mapsto 0, y \mapsto 2)\rangle
\end{array}
$$

Figure 3.29: Sample reduction sequence.

IMP evaluation contexts syntax	IMP language syntax
Context ::= □ | *Context* + *AExp* | *AExp* + *Context* | *Context* / *AExp* | *AExp* / *Context* | *Context* <= *AExp* | *Int* <= *Cxt* | not *Context* | *Context* and *BExp* | *Id* := *Context* | *Context* ; *Stmt* | if *Context* then *Stmt* else *Stmt*	*AExp* ::= *Int* | *Id* | | *AExp* + *AExp* | *AExp* / *AExp* *BExp* ::= *Bool* | *AExp* <= *AExp* | not *BExp* | *BExp* and *BExp* *Stmt* ::= *Id* := *AExp* | *Stmt* ; *Stmt* | if *BExp* then *Stmt* else *Stmt* | while *BExp* do *Stmt* *Pgm* ::= var **List**{*Id*} ; *Stmt*

Figure 3.30: Evaluation contexts for IMP (left column); the syntax of IMP (from Figure 3.1) is recalled in the right column only for reader's convenience, to more easily compare the two grammars.

3.7.1 The Reduction Semantics with Evaluation Contexts of IMP

Figure 3.30 shows the definition of evaluation contexts for IMP and Figure 3.31 shows all the reduction semantics rules of IMP using the evaluation contexts defined in Figure 3.30. The evaluation context productions capture the intended evaluation strategies of the various language constructs. For example, `+` and `/` are non-deterministically strict, so any one of their arguments can be reduced one step whenever the sum or the division expression can be reduced one step, respectively, so the hole $\square$ can go in any of their two subexpressions. As previously discussed, in the case of `<=` one can reduce its second argument only after its first argument is fully reduced (to an integer). The evaluation strategy of `not` is straightforward. For `and`, note that only its first argument is reduced. Indeed, recall that `and` has a short-circuited semantics, so its second argument is reduced only after the first one is completely reduced (to a Boolean) and only if needed; this is defined using rules in Figure 3.31. The evaluation contexts for assignment, sequential composition, and the conditional have already been discussed.

Many of the rules in Figure 3.31 have already been discussed or are trivial and thus deserve no discussion. Note that there is no production *Context* ::= `while` *Context* `do` *Stmt* as a hasty reader may (mistakenly) expect. That is because such a production would allow the evaluation of the Boolean expression in the while loop's condition to a Boolean value in the current context; supposing that value is `true`, then, unless one modifies the syntax in some rather awkward way, there is no chance to recover the original Boolean expression to evaluate it again after the evaluation of the while loop's body statement. The solution to handle loops remains the same as in SOS, namely to explicitly unroll them into conditional statements, as shown in Figure 3.31. Note that the evaluation contexts allow the loop unrolling to only happen when the `while` statement is a redex. In particular, after an unrolling reduction takes place, subsequent unrolling steps are disallowed inside the `then` branch; to unroll it again, the loop statement must become again a redex, which can only happen after the conditional statement is itself reduced. The initial program configuration (containing only the program) is reduced also like in SOS (last rule in Figure 3.31; note that one cannot instead define a production *Context* ::= `var` **List**`{`*Id*`}` `;` *Context*, because, for example, there is no way to reduce the statement x `:=` 5 in $\langle$ `var` x `;` $\boxed{x \texttt{ := } 5}\rangle$.

3.7.2 The Reduction Semantics with Evaluation Contexts of IMP++

We next discuss the reduction semantics of IMP++ using evaluation contexts. Like for the other semantics, we first add each feature separately to IMP and then we add all of them together and investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The use of evaluation contexts makes the definition of variable increment quite elegant and modular:

$$\langle c, \sigma \rangle [\texttt{++}\, x] \quad \rightarrow \quad \langle c, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle [\sigma(x) +_{Int} 1]$$

No other rule needs to change, because: (1) unlike in SOS, all the language-specific rules are unconditional, each rule matching and modifying only its relevant part of the configuration; and (2) the language-independent characteristic rule allows reductions to match and modify only their relevant part of the configuration, propagating everything else in the configuration automatically.

$$Context ::= \ldots \mid \langle Context, State \rangle$$

$$\frac{e \to e'}{c[e] \to c[e']}$$

$$\begin{array}{rcll}
\langle c, \sigma \rangle[x] & \to & \langle c, \sigma \rangle[\sigma(x)] & \text{if } \sigma(x) \neq \bot \\
i_1 \texttt{ + } i_2 & \to & i_1 +_{Int} i_2 & \\
i_1 \texttt{ / } i_2 & \to & i_1 /_{Int} i_2 & \text{if } i_2 \neq 0 \\
i_1 \texttt{ <= } i_2 & \to & i_1 \leq_{Int} i_2 & \\
\texttt{not true} & \to & \texttt{false} & \\
\texttt{not false} & \to & \texttt{true} & \\
\texttt{true and } b_2 & \to & b_2 & \\
\texttt{false and } b_2 & \to & \texttt{false} & \\
\langle c, \sigma \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x] \rangle[\texttt{ skip }] & \text{if } \sigma(x) \neq \bot \\
\texttt{skip ; } s_2 & \to & s_2 & \\
\texttt{if true then } s_1 \texttt{ else } s_2 & \to & s_1 & \\
\texttt{if false then } s_1 \texttt{ else } s_2 & \to & s_2 & \\
\texttt{while } b \texttt{ do } s & \to & \texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else skip} & \\
\langle \texttt{ var } xl \texttt{ ; } s \rangle & \to & \langle s, (xl \mapsto 0) \rangle &
\end{array}$$

Figure 3.31: RSEC(IMP): The reduction semantics with evaluation contexts of IMP ($e, e' \in AExp \cup BExp \cup Stmt$; $c \in Context$ appropriate (that is, the respective terms involving c are well-formed); $i, i_1, i_2 \in Int$; $x \in Id$; $b, b_2 \in BExp$; $s, s_1, s_2 \in Stmt$; $xl \in \mathbf{List}\{Id\}$; $\sigma \in State$).

Input/Output

We need to first change the configuration employed by our reduction semantics with evaluation contexts of IMP from $\langle s, \sigma \rangle$ to $\langle s, \sigma, \omega_{in}, \omega_{out} \rangle$, to also include the input/output buffers. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, we need to change the syntax of (statement) configurations to include input and output buffers, and the syntax of contexts from $Context ::= \ldots \mid \langle Context, State \rangle$ to $Context ::= \ldots \mid \langle Context, State, Buffer, Buffer \rangle$. No matter what semantic approach one employs, some changes in the configuration (or its equivalent) are unavoidable when one adds new language features that require new semantic data, like input/output constructs that require their own buffers (recall that, e.g., in MSOS, we had to add new attributes in transition labels instead). Hence, this change is acceptable. What is inconvenient (and non-modular), however, is that the rules for variable lookup and for assignment need the complete configuration, so they have to change from

$$\begin{array}{rcl}
\langle c, \sigma \rangle[x] & \to & \langle c, \sigma \rangle[\sigma(x)] \\
\langle c, \sigma \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x] \rangle[\texttt{skip}]
\end{array}$$

to

$$\begin{array}{rcl}
\langle c, \sigma, \omega_{in}, \omega_{out} \rangle[x] & \to & \langle c, \sigma, \omega_{in}, \omega_{out} \rangle[\sigma(x)] \\
\langle c, \sigma, \omega_{in}, \omega_{out} \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x], \omega_{in}, \omega_{out} \rangle[\texttt{skip}]
\end{array}$$

Also, the initial configuration which previously held only the program, now has to change to hold both the program and an input buffer, and the rule for programs (the last one in Figure 3.31) needs

to change as follows:

$$\langle \texttt{var}\, xl\, \texttt{;}\, s, \omega_{in} \rangle \rightarrow \langle s, (xl \mapsto 0), \omega_{in}, \epsilon \rangle$$

Once the changes above are applied, we are ready for adding the evaluation context for the print statement as well as the reduction semantics rules of both input/output constructs:

$$\begin{array}{rcl} \mathit{Context} & ::= & \ldots \mid \texttt{print(}\, \mathit{Context} \texttt{)} \\ \langle c, \sigma, i : \omega_{in}, \omega_{out} \rangle [\texttt{read()}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out} \rangle [i] \\ \langle c, \sigma, \omega_{in}, \omega_{out} \rangle [\texttt{print(}\, i \texttt{)}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out} : i \rangle [\texttt{skip}] \end{array}$$

Other possibilities to add input/output buffers to the configuration and to give the reduction semantics with evaluation contexts of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.10.

Abrupt Termination

For our language, reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language and without adding any new reduction steps as an artifact of the approach chosen:

$$\begin{array}{rcl} \langle c, \sigma \rangle [i \,/\, 0] & \rightarrow & \langle\, \texttt{skip}\, , \sigma \rangle \\ \langle c, \sigma \rangle [\,\texttt{halt}\,] & \rightarrow & \langle\, \texttt{skip}\, , \sigma \rangle \end{array}$$

Therefore, the particular evaluation context in which the abrupt termination is being generated, c, is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in those styles the evaluation context c is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggest that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

Dynamic Threads

The rules (SmallStep-Spawn-Arg) (resp. (MSOS-Spawn-Arg)) and (SmallStep-Spawn-Wait) (resp. (MSOS-Spawn-Wait)) in Section 3.5.4 (resp. Section 3.6.2) are essentially computation propagation rules. In reduction semantics with evaluation contexts the role of such rules is taken over by the splitting/plugging mechanism, which in turn relies on parsing and therefore needs productions for evaluation contexts. We can therefore replace those rules by appropriate productions for evaluation contexts:

$$\begin{array}{rcl} \mathit{Context} & ::= & \ldots \\ & \mid & \texttt{spawn(}\, \mathit{Context} \texttt{)} \\ & \mid & \texttt{spawn(}\, \mathit{Stmt} \texttt{)}\, \texttt{;}\, \mathit{Context} \end{array}$$

The second evaluation context production above involves two language constructs, namely `spawn` and sequential composition. The desired non-determinism due to concurrency is captured by deliberate ambiguity in parsing evaluation contexts and, implicitly, in the splitting/plugging mechanism.

The remaining rule (SmallStep-Spawn-Skip) (resp. (MSOS-Spawn-Skip)) in Section 3.5.4 (resp. Section 3.6.2) is turned into an equivalent rule here, and the structural identity stating the associativity of sequential composition is also still necessary:

$$\begin{array}{rcl} \texttt{spawn skip} & \rightarrow & \texttt{skip} \\ (s_1\, \texttt{;}\, s_2)\, \texttt{;}\, s_3 & \equiv & s_1\, \texttt{;}\, (s_2\, \texttt{;}\, s_3) \end{array}$$

Local Variables

We make use of the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs, to reduce the problem to only give semantics to `let`. Recall from Section 3.5.5 that the semantics of $\texttt{let}\, x = a\, \texttt{in}\, s$ in a state σ is to first evaluate arithmetic expression a in σ to some integer i and then evaluate statement s in state $\sigma[i/x]$; after the evaluation of s, the value of x is recovered to $\sigma(x)$ (i.e., whatever it was before the execution of the block) but all the other state updates produced by the evaluation of s are kept. These suggest the following:

$$\textit{Context} \;::=\; \ldots \;\mid\; \texttt{let}\; \textit{Id} = \textit{Context}\; \texttt{in}\; \textit{Stmt}$$

$$\langle c, \sigma\rangle[\texttt{let}\, x = i\, \texttt{in}\, s] \;\rightarrow\; \langle c, \sigma[i/x]\rangle[s\, ;\, x := \sigma(x)]$$

Notice that a solution similar to that in small-step SOS and MSOS (see rules (SmallStep-Let-Stmt) and (MSOS-Let-Stmt) in Sections 3.5.5 and 3.6.2, respectively) does not work here, because rules in reduction semantics with evaluation contexts are unconditional. In fact, a solution similar to the one we adopted above was already discussed in Section 3.5.6 in the context of small-step SOS, where we also explained that it works as shown because of a syntactic trick, namely because we allow assignment statements of the form $x := \bot$ (in our case here when $\sigma(x) = \bot$), which have the effect to undefine σ in x (see Section 2.3.2). Section 3.5.6 also gives suggestions on how to avoid allowing $\bot$ to be assigned to x, if one does not like it. One suggestion was to rename the bound variable into a fresh one, this way relieving us from having to recover its value after the `let`:

$$\langle c, \sigma\rangle[\texttt{let}\, x = i\, \texttt{in}\, s] \;\rightarrow\; \langle c, \sigma[i/x']\rangle[s[x'/x]] \qquad \text{where } x' \text{ fresh}$$

This approach comes with several problems, though: it requires that we define and maintain a substitution operation (for $s[x'/x]$), we have to pay a complexity linear with the size of the `let` body each time a `let` statement is eliminated, and the state may grow indefinitely (since the `let` can be inside a loop).

Note also that, with the current reduction semantics with evaluation contexts of IMP, we cannot add the following evaluation context production

$$\textit{Context} \;::=\; \ldots \;\mid\; \texttt{let}\; \textit{Id} = \textit{Int}\; \texttt{in}\; \textit{Context}$$

stating that once the binding expression becomes an integer then we can evaluate the `let` body. We cannot add it simply because we have to bind the variable to the integer before we evaluate the `let` body statement. An evaluation context production like above would result in evaluating the `let` body statement in the same state as before the `let`, which is obviously wrong. However, the existence of a `let` binder allows us to possibly rethink the overall reduction semantics of IMP, to make it more syntactic. Indeed, the `let` binders can be used in a nested manner and hereby allow us to syntactically mimic a state. For example, a statement of the form $\texttt{let}\, x = 5\, \texttt{in}\, \texttt{let}\, y = 7\, \texttt{in}\, s$ can be thought of as the statement s being executed in a "state" where x is bound to 5 and where y is bound to 7. We can then drop the configurations of IMP completely and instead add the evaluation context above allowing reductions inside `let` body statements. The IMP rules that refer to configurations, namely those for lookup and assignment, need to change to work with the new "state", and a new rule to eliminate unnecessary `let` statements needs to be added:

$$\begin{array}{rcl} \texttt{let}\, x = i\, \texttt{in}\, c[x] & \rightarrow & \texttt{let}\, x = i\, \texttt{in}\, c[i] \\ \texttt{let}\, x = i\, \texttt{in}\, c[x := j] & \rightarrow & \texttt{let}\, x = j\, \texttt{in}\, c[\texttt{skip}] \\ \texttt{let}\, x = i\, \texttt{in}\, \texttt{skip} & \rightarrow & \texttt{skip} \end{array}$$

The above works correctly only if one ensures that the evaluation contexts c do not contain other $\texttt{let}\, x = \_\, \texttt{in}\, \_$ evaluation context constructs, with the same x variable name as in the rules (the underscores can be any integer and context, respectively). One can do this by statically renaming the bound variables to have different names at parse-time, or by employing a substitution operation to do it dynamically. Note that the static renaming approach requires extra-care as the language is extended, particularly if new `let` statements can be generated dynamically by other language constructs. Another approach to ensure the correct application of the rules above, which is theoretically more complex and practically more expensive and harder to implement, is to add a side condition to the first two rules of the form "where c does not contain any evaluation context production instance of the form $\texttt{let}\, x = \_\, \texttt{in}\, \_$".

To conclude, the discussion above suggests that there are various ways to give a reduction semantics of `let` using evaluation contexts, none of them absolutely better than the others: some are simpler, others are more syntactic but require special external support, others require non-modular changes to the existing language. The approaches above are by no means exhaustive. For example, we have not even discussed environment-store based approaches.

Putting Them All Together

It is relatively easy to combine all the reduction semantics with evaluation contexts of all the features above, although not as modularly as it was for MSOS. Specifically, we have to do the following:

1. Apply all the changes that we applied when we added input/output to IMP above, namely: add input/output buffers to both configurations and configuration evaluation contexts; change the semantic rules involving configurations to work with the extended configurations, more precisely the rules for variable lookup, for variable assignment, and for programs.

2. Add all the evaluation contexts and the rules for the individual features above, making sure we *change* those of them using configurations or configuration evaluation contexts (i.e., almost all of them) to work with the new configurations including input/output buffers.

Unfortunately, the above is not giving us the desired language. Worse, it actually gives us a wrong language, namely one with a disastrous feature interaction. This problem has already been noted in Section 3.5.6, where we discussed the effect of a similar semantics to that of `let` above, but using small-step SOS instead of evaluation contexts. The problem here is that, if the body of a `let` statement contains a `spawn` statement, then the latter will be allowed, according to its semantics, to be executed in parallel with the statements following it. In our case, the assignment statement $x := \sigma(x)$ in the `let` semantics, originally intended to recover the value of x, can be now potentially executed before the spawned statement, resulting in a wrong behavior; in particular, the assignment can even "undefine" x in case $\sigma(x) = \bot$, in which case the `spawn` statement can even get stuck.

As already indicated in Section 3.5.6, the correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`'s body statement, this way eliminating the need to recover the value of the bound variable to what it was before the `let`:

$$\langle c, \sigma, \omega_{in}, \omega_{out}\rangle[\texttt{let}\, x = i\, \texttt{in}\, s] \;\rightarrow\; \langle c, \sigma[i/x'], \omega_{in}, \omega_{out}\rangle[s[x'/x]] \qquad \text{where } x' \text{ is a fresh variable}$$

We have used configurations already extended with input/output buffers, as needed for IMP++. This solution completely brakes any (intended or unintended) relationship between the `let` construct and any other language constructs that may be used inside its body, although, as discussed in Section 3.5.6, the use of the substitution comes with a few (relatively acceptable) drawbacks.

sort:
$Syntax$ // includes all syntactic terms, in contextual representation $context[redex]$ or not
subsorts:
$N_1, N_2, \ldots < Syntax$ // N_1, N_2, ..., are sorts whose terms can be regarded as $context[redex]$
operations:
$\_[\_] : Context \times Syntax \to Syntax$ // constructor for terms in contextual representation
$split : Syntax \to Syntax$ // puts syntactic terms into contextual representation
$plug : Syntax \to Syntax$ // the dual of split
rules and equations:
$split(Syn) \to \Box[Syn]$ // generic rule; it initiates the splitting process for the rules below
$plug(\Box[Syn]) = Syn$ // generic equation; it terminates the plugging process
// for each context production $Context ::= \pi(N_1, \ldots, N_n, Context)$ add the following:
$split(\pi(T_1, \ldots, T_n, T)) \to \pi(T_1, \ldots, T_n, C)[Syn]$ **if** $split(T) \to C[Syn]$
$plug(\pi(T_1, \ldots, T_n, C)[Syn]) = \pi(T_1, \ldots, T_n, plug(C[Syn]))$

Figure 3.32: Embedding evaluation contexts into rewriting logic theory $\mathcal{R}^{\Box}_{\mathrm{RSEC}}$. The implicit split/plug mechanism is replaced by explicit rewriting logic sentences achieving the same task (the involved variables have the sorts $Syn : Syntax$, $C : Context$, $T_1 : N_1$, ..., $T_n : N_n$, and $T : N$).

3.7.3 Reduction Semantics with Evaluation Contexts in Rewriting Logic

In this section we show how to automatically and faithfully embed reduction semantics with evaluation contexts into rewriting logic. After discussing how to embed evaluation contexts into rewriting logic, we first give a straightforward embedding of reduction semantics, which is easy to prove correct but which does not take advantage of performance-improving techniques currently supported by rewrite engines, so consequently it is relatively inefficient when executed or formally analyzed. We then discuss simple optimizations which increase the performance of the resulting rewrite definitions an order of magnitude or more. We only consider evaluation contexts which can be defined by means of context-free grammars (CFGs). However, the CFG that we allow for defining evaluation contexts can be non-deterministic, in the sense that a term is allowed to split many different ways into a context and a redex (like the CFG in Figure 3.30).

Faithful Embedding of Evaluation Contexts into Rewriting Logic

Our approach to embedding reduction semantics with evaluation contexts in rewriting logic builds on an embedding of evaluation contexts and their implicit splitting/plugging mechanism in rewriting logic. More precisely, each evaluation context production is associated with an equation (for plugging) and a conditional rewrite rule (for splitting). The conditional rewrite rules allow to non-deterministically split a term into a context and a redex. Moreover, when executing the resulting rewriting logic theory, the conditional rules allow for finding *all* splits of a term into a context and a redex, provided that the underlying rewrite engine has search capabilities (like Maude does).

Figure 3.32 shows a general and automatic procedure to generate a rewriting logic theory from any CFG defining evaluation contexts for some given language syntax. Recall that, for simplicity, in this section we assume only one *Context* syntactic category. What links the CFG of evaluation contexts to the CFG of the language to be given a semantics, which is also what makes our embedding

into rewriting logic discussed here work, is the assumption that for any context production

$$Context \ ::= \ \pi(N_1, \ldots, N_n, Context)$$

there are some syntactic categories N, N' (different or not) in the language CFG (possibly extended with configurations and semantic components as discussed above) such that $\pi(t_1, \ldots, t_n, t) \in N'$ for any $t_1 \in N_1$, ..., $t_n \in N_n$, $t \in N$. We here used a notation which needs to be explained. The actual production above is $Context ::= \pi$, where π is a string of terminals and non-terminals, but we write $\pi(N_1, \ldots, N_n, Context)$ instead of π to emphasize that $N_1, \ldots, N_n, Context$ are all the non-terminals appearing in π; we listed the *Context* last for simplicity. Also, by abuse of notation, we let $\pi(t_1, \ldots, t_n, t)$ denote the term obtained by substituting $(t_1, \ldots, t_n, t)$ for $(N_1, \ldots, N_n, Context)$ in π, respectively. So our assumption is that $\pi(t_1, \ldots, t_n, t)$ is well-formed under the syntax of the language whenever t_1, ..., t_n, t are well-defined in the appropriate syntactic categories, that is, $t_1 \in N_1$, ..., $t_n \in N_n$, $t \in T$. This is indeed a very natural property of well-defined evaluation contexts for a given language, so natural that one may even ask how it can be otherwise (it is easy to violate this property though, e.g., $Context ::= \texttt{<=} Context \texttt{ := } AExp$). Without this property, our embedding of evaluation contexts in rewriting logic in Figure 3.32 would not be well-formed, because the left-hand side terms of some of the conditional rule(s) for split would not be well-formed terms.

For simplicity, in Figure 3.32 we prefer to subsort all the syntactic categories whose terms are intended to be allowed contextual representations $context[redex]$ under one top sort[6], *Syntax*. The implicit notation $context[term]$ for contextual representations, as well as the implicitly assumed *split* and *plug* operations, are defined explicitly in the corresponding rewrite theory. The split operation is only defined on terms over the original language syntax, while the plug operation is defined only over terms in contextual representation. One generic (i.e., independent of the particular RSEC definition) rule and one generic equation are added: $split(Syn) \to \Box[Syn]$ initiates the process of splitting a term into a contextual representation and $plug(\Box[Syn]) = Syn$ terminates the process of plugging a term into a context. It is important that the first be a rewrite rule (because it can lead to non-determinism; this is explained below), while the second can safely be an equation.

Each evaluation context production translates into one equation and one conditional rewrite rule. The equation tells how terms are plugged into contexts formed with that production, while the conditional rule tells how that production can be used to split a term into a context and a redex. The equations defining plugging are straightforward: for each production in the original CFG of evaluation contexts, iteratively plug the subterm in the smaller context; when the hole is reached, replace it by the subterm via the generic equation. The conditional rules for splitting also look straightforward, but how and why they work is more subtle. For any context production, if the term to split matches the pattern of the production, then first split the subterm corresponding to the position of the subcontext and then use that contextual representation of the subterm to construct the contextual representation of the original term; at any moment, one has the option to stop splitting thanks to the generic rule $split(Syn) \to \Box[Syn]$. For example, for the five *Context*

[6] An alternative, which does not involve subsorting, is to replace each syntactic category as above by *Syntax*. Our construction also works without subsorting and without collapsing of syntactic categories, but it is more technical, requires more operations, rules, and equations, and it is likely not worth the effort without a real motivation to use it in term rewrite settings without support for subsorting. We have made experiments with both approaches and found no penalty on performance when collapsing syntactic categories.

productions in the evaluation context CFG in the preamble of this section, namely

$$\begin{array}{rcl} \mathit{Context} & ::= & \mathit{Context} \texttt{ <= } \mathit{AExp} \\ & | & \mathit{Int} \texttt{ <= } \mathit{Context} \\ & | & \mathit{Id} \texttt{ := } \mathit{Context} \\ & | & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\ & | & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \end{array}$$

the general procedure in the rewriting logic embedding of evaluation contexts in Figure 3.32 yields the following five rules and five equations (variable I_1 has sort *Int*; X has sort *Id*; A_1 and A_2 have sort *AExp*; B has sort *BExp*; S_1 and S_2 have sort *Stmt*; C has sort *Context*; *Syn* has sort *Syntax*):

$$\begin{array}{l} \mathit{split}(A_1 \texttt{ <= } A_2) \to (C \texttt{ <= } A_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(A_1) \to C[\mathit{Syn}] \\ \mathit{plug}((C \texttt{ <= } A_2)[\mathit{Syn}]) = \mathit{plug}(C[\mathit{Syn}]) \texttt{ <= } A_2 \end{array}$$

$$\begin{array}{l} \mathit{split}(I_1 \texttt{ <= } A_2) \to (I_1 \texttt{ <= } C)[\mathit{Syn}] \textbf{ if } \mathit{split}(A_2) \to C[\mathit{Syn}] \\ \mathit{plug}((I_1 \texttt{ <= } C)[\mathit{Syn}]) = I_1 \texttt{ <= } \mathit{plug}(C[\mathit{Syn}]) \end{array}$$

$$\begin{array}{l} \mathit{split}(X \texttt{ := } A) \to (X \texttt{ := } C)[\mathit{Syn}] \textbf{ if } \mathit{split}(A) \to C[\mathit{Syn}] \\ \mathit{plug}((X \texttt{ := } C)[\mathit{Syn}]) = X \texttt{ := } \mathit{plug}(C[\mathit{Syn}]) \end{array}$$

$$\begin{array}{l} \mathit{split}(S_1 \texttt{ ; } S_2) \to (C \texttt{ ; } S_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(S_1) \to C[\mathit{Syn}] \\ \mathit{plug}((C \texttt{ ; } S_2)[\mathit{Syn}]) = \mathit{plug}(C[\mathit{Syn}]) \texttt{ ; } S_2 \end{array}$$

$$\begin{array}{l} \mathit{split}(\texttt{ if } B \texttt{ then } S_1 \texttt{ else } S_2) \to (\texttt{ if } C \texttt{ then } S_1 \texttt{ else } S_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(B) \to C[\mathit{Syn}] \\ \mathit{plug}((\texttt{ if } C \texttt{ then } S_1 \texttt{ else } S_2)[\mathit{Syn}]) = \texttt{ if } \mathit{plug}(C[\mathit{Syn}]) \texttt{ then } S_1 \texttt{ else } S_2 \end{array}$$

The reason for which we used rules instead of equations for splitting in our embedding in Figure 3.32 is that splitting, unlike plugging, can be non-deterministic. Recall that the use of an arrow/transition in the condition of a rule has an existential nature (Sections 2.7). In particular, rewriting logic engines should execute conditional rules by performing an exhaustive search, or reachability analysis of all the zero-, one- or more-step rewrites of the condition left-hand side term ($\mathit{split}(T)$ in our case) into the condition right-hand side term ($C[\mathit{Syn}]$ in our case). Such an exhaustive search explores all possible splits, or parsings, of a term into a contextual representation.

Theorem 6. ***(Embedding splitting/plugging into rewriting logic)*** *Given an evaluation context CFG as discussed above, say as part of some reduction semantics with evaluation contexts definition* RSEC*, let* $\mathcal{R}^{\Box}_{\text{RSEC}}$ *be the rewriting logic theory associated to it as in Figure 3.32. Then the following are equivalent for any* $t, r \in \mathit{Syntax}$ *and* $c \in \mathit{Context}$*:*

- t *can be split as* $c[r]$ *using the evaluation context CFG of* RSEC*;*
- $\mathcal{R}^{\Box}_{\text{RSEC}} \vdash \mathit{split}(t) \to c[r]$*;*
- $\mathcal{R}^{\Box}_{\text{RSEC}} \vdash \mathit{plug}(c[r]) = t$*.*

The theorem above says that the process of splitting a term t into a context and a redex in reduction semantics with evaluation contexts, which can be non-deterministic, reduces to reachability in the corresponding rewriting logic theory of a contextual representation pattern $c[r]$ of the original term marked for splitting, $\mathit{split}(t)$. Rewrite engines such as Maude provide a search command that does precisely that. We will shortly see how Maude's search command can find all splits of a term.

rules:
// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following conditional semantic rewrite rule:

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to \overline{r}(plug(\overline{c'_1}[\overline{r_1}]), \ldots, plug(\overline{c'_n}[\overline{r_{n'}}])) \text{ if } split(T_1) \to \overline{c_1}[\overline{l_1}] \wedge \ \ldots \ \wedge split(T_n) \to \overline{c_n}[\overline{l_n}]$$

Figure 3.33: First embedding of RSEC into rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{1}}_{\text{RSEC}}$).

Faithful Embedding of RSEC in Rewriting Logic

In this section we discuss three faithful rewriting logic embeddings of reduction semantics with evaluation contexts. The first two assume that the embedded reduction semantics has no characteristic rule, in that all reductions take place at the top of the original term to reduce (e.g., a configuration in the case of our IMP language); this is not a limitation because, as already discussed, the characteristic rule can be regarded as syntactic sugar anyway, its role being to allow one to write reduction semantics definitions more compactly and elegantly. The first embedding is the simplest and easiest to prove correct, but it is the heaviest in notation and the resulting rewrite theories tend to be inefficient when executed because most of the left-hand-side terms of rules end up being identical, thus making the task of matching and selecting a rule to apply rather complex for rewrite engines. The second embedding results in rewrite rules whose left-hand-side terms are mostly distinct, thus taking advantage of current strengths of rewrite engines to index terms so that rules to be applied can be searched for quickly. Our third embedding is as close to the original reduction semantics in form and shape as one can hope it to be in a rewriting setting; in particular, it also defines a characteristic rule, which can be used to write a more compact semantics. The third embedding yields rewrite theories which are as efficient as those produced by the second embedding. The reason we did not define directly the third embedding is because we believe that the transition from the first to the second and then to the third is instructive.

Since reduction semantics with evaluation contexts is an inherently small-step semantical approach, we use the same mechanism to control the rewriting as for small-step SOS (Section 3.3) and MSOS (Section 3.6). This mechanism was discussed in detail in Section 3.3.3. It essentially consists of: (1) tagging each left-hand-side term appearing in a rule transition with a $\circ$, to capture the desired notion of a one-step reduction of that term; and (2) tagging with a $\star$ the terms to be multi-step (zero, one or more steps) reduced, where $\star$ can be easily defined with a conditional rule as the transitive and reflexive closure of $\circ$ (see Section 3.3.3).

Figure 3.33 shows our first embedding of reduction semantics with evaluation contexts into rewriting logic, which assumes that the characteristic rule, if any, has already been desugared. Each reduction semantics rule translates into one conditional rewrite rule. We allow the reduction rules to have in their left-hand-side and right-hand-side terms an arbitrary number of subterms that are in contextual representation. For example, if the left-hand side l of a reduction rule has n such subterms, say $c_1[l_1]$, ..., $c_n[l_n]$, then we write it $l(c_1[l_1], \ldots, c_n[l_n])$ (this is similar with our previous notation $\pi(N_1, \ldots, N_n, N)$ in the section above on embedding of evaluation contexts into rewriting logic, except that we now single out all the subterms in contextual representation instead of all the non-terminals). In particular, a rule $l \to r$ in which l and r contain no subterms in contextual representation (like the last rule in Figure 3.31) is translated exactly like in small-step SOS, that is,

into $\overline{l} \to \overline{r}$. Also, note that we allow evaluation contexts to have any pattern (since we overline them, like any other terms); we do not restrict them to only be context variables. Consider, for example, the six reduction rules discussed in the preamble of Section 3.7, which after the desugaring of the characteristic rule are as follows:

$$\begin{aligned}
c[i_1 \texttt{ <= } i_2] &\to c[i_1 \leq_{Int} i_2] \\
c[\texttt{ skip ; } s_2] &\to c[s_2] \\
c[\texttt{ if true then } s_1 \texttt{ else } s_2] &\to c[s_1] \\
c[\texttt{ if false then } s_1 \texttt{ else } s_2] &\to c[s_2] \\
\langle c, \sigma \rangle[x] &\to \langle c, \sigma \rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
\langle c, \sigma \rangle[x \texttt{ := } i] &\to \langle c, \sigma[i/x] \rangle[\texttt{ skip }] \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

Since all these rules have left-hand side-terms already in contextual representation, their corresponding l in Figure 3.33 is just a non-terminal (*Configuration*), which means that $\overline{l}$ is just a variable (of sort *Configuration*). Therefore, the rewriting logic rules associated to these RSEC rules are:

$$\begin{aligned}
&\circ \mathit{Cfg} \to \mathit{plug}(C[I_1 \leq_{Int} I_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[I_1 \texttt{ <= } I_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ skip ; } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_1]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ if true then } S_1 \texttt{ else } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ if false then } S_1 \texttt{ else } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(\langle C, \sigma \rangle[\sigma(X)]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to \langle C, \sigma \rangle[X] \ \wedge \ \sigma(X) \neq \bot \\
&\circ \mathit{Cfg} \to \mathit{plug}(\langle C, \sigma[I/X] \rangle[\texttt{ skip }]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to \langle C, \sigma \rangle[X \texttt{ := } I] \ \wedge \ \sigma(X) \neq \bot
\end{aligned}$$

Recall from the preamble of Section 3.7 that the RSEC rules for variable lookup and assignment can also be given as follows, so that their left-hand-side terms are not in contextual representation:

$$\begin{aligned}
\langle c[x], \sigma \rangle &\to \langle c[\sigma(x)], \sigma \rangle \quad \text{if } \sigma(x) \neq \bot \\
\langle c[x \texttt{ := } i], \sigma \rangle &\to \langle c[\texttt{ skip }], \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

In these cases, the string l in Figure 3.33 has the form $\langle \mathit{Stmt}, \sigma \rangle$, which means that $\overline{l}$ is the term $\langle S, \sigma \rangle$, where S is a variable of sort *Stmt*. Then their corresponding rewriting logic rules are:

$$\begin{aligned}
&\circ \langle S, \sigma \rangle \to \langle \mathit{plug}(C[\sigma(X)]), \sigma \rangle \textbf{ if } \mathit{split}(S) \to C[X] \ \wedge \ \sigma(X) \neq \bot \\
&\circ \langle S, \sigma \rangle \to \langle \mathit{plug}(C[\texttt{ skip }]), \sigma[I/X] \rangle \textbf{ if } \mathit{split}(S) \to C[X \texttt{ := } I] \ \wedge \ \sigma(X) \neq \bot
\end{aligned}$$

Once the characteristic rule is desugared as explained in the preamble of Section 3.7, an RSEC rule operates as follows: (1) attempt to match the left-hand-side pattern of the rule at the top of the term to reduce, making sure that each of the subterms corresponding to subpatterns in contextual representation form can indeed be split as indicated; and (2) if the matching step above succeeds, then reduce the original term to the right-hand-side pattern instantiated accordingly, plugging all the subterms appearing in contextual representations in the right-hand side. Note that the conditional rewrite rule associated to an RSEC rule as indicated in Figure 3.33 achieves precisely the desired steps above: the $\circ$ in the left-hand-side term guarantees that the rewrite step takes place at the top of the original term, the condition exhaustively searches for the desired splits of the subterms in question into contextual representations, and the right-hand side plugs back all the contextual representations into terms over the original syntax. The only difference between the original RSEC rule and its corresponding rewriting logic conditional rule is that the rewriting logic rule makes explicit the splits and plugs that are implicit in the RSEC rule.

Theorem 7. ***(First faithful embedding of reduction semantics into rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.33. Then*

1. ***(step-for-step correspondence)*** $\text{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \circ\, \overline{t} \to^1 \overline{t'}$ *using the corresponding conditional rewrite rule obtained like in Figure 3.33; moreover, the reduction rule and the corresponding rewrite rule apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 6);*

2. ***(computational correspondence)*** $\text{RSEC} \vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \star\, \overline{t} \to \star\, \overline{t'}$.

The first item in Theorem 7 says that the resulting rewriting logic theory captures faithfully the small-step reduction relation of the original reduction semantics with evaluation contexts definition. The faithfulness of this embedding (i.e., there is precisely one top-level application of a rewrite rule that corresponds to an application of a reduction semantics rule), comes from the fact that the consistent use of the $\circ$ tag inhibits any other application of any other rule on the tagged term. Therefore, like in small-step SOS and MSOS, a small-step in a reduction semantics definition also reduces to reachability analysis in the corresponding rewrite theory; one can also use the search capability of a system like Maude to find all the next terms that a given term evaluates to (Maude provides the capability to search for the first n terms that match a given pattern using up to m rule applications, where n and m are user-provided parameters).

The step-for-step correspondence above is stronger (and better) than the strong bisimilarity of the two definitions; for example, if a reduction semantics rule in RSEC can be applied in two different ways on a term to reduce, then its corresponding rewrite rule in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ can also be applied in two different ways on the tagged term. The second item in Theorem 7 says that the resulting rewrite theory can be used to perform any computation possible in the original RSEC, and vice versa (the step-for-step correspondence is guaranteed in combination with the first item). Therefore, there is absolutely no difference between computations using RSEC and computations using $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$, except for irrelevant syntactic conventions/notations. This strong correspondence between reductions in RSEC and rewrites in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ tells that $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *is precisely* RSEC*, not an encoding of it*. In other words, RSEC can be faithfully regarded as a methodological fragment of rewriting logic, same like big-step SOS, small-step SOS, and MSOS.

The discussion above implies that, from a theoretical perspective, the rewriting logic embedding of reduction semantics in Figure 3.33 is as good as one can hope. However, its simplicity comes at a price in performance, which unfortunately tends to be at its worst precisely in the most common cases. Consider, for example, the six rewrite rules used before Theorem 7 to exemplify the embedding in Figure 3.33 (consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top—first variant). They all have the form:

$$\circ\, \mathit{Cfg} \to \ldots \;\; \textbf{if} \;\; \mathit{split}(\mathit{Cfg}) \to \ldots$$

In fact, as seen in Figure 3.38, all the rewrite rules in the rewriting logic theory corresponding to the RSEC of IMP have the same form. The reason the left-hand-side terms of these rewrite rules are the same and lack any structure is because the contextual representations in the left-hand-side terms of the RSEC rules appear at the top, with no structure above them, which is the most common type of RSEC rule encountered.

To apply a conditional rewrite rule, a rewrite engine first matches the left-hand side and then performs the (exhaustive) search in the condition. In other words, the structure of the left-hand side acts as a cheap guard for the expensive search. Unfortunately, since the left-hand side of the conditional rewrite rules above has no structure, it will always match. That means that the searches in the conditions of all the rewrite rules will be, in the worst case, executed one after another until a split is eventually found (if any). If one thinks in terms of implementing RSEC in general, then this is what a naive implementation would do. If one thinks in terms of executing term rewrite systems, then this fails to take advantage of some important performance-increasing advances in term rewriting, such as *indexing* [82, 83, 3]. In short, indexing techniques use the structure of the left-hand sides to augment the term structure with information about which rule can potentially be applied at which places. This information is dynamically updated, as the term is rewritten. If the rules' left-hand sides do not significantly overlap, it is generally assumed that it takes constant time to find a matching rewrite rule. This is similar in spirit to hashing, where the access time into a hash table is generally assumed to take constant time when there are no or few key collisions. Thinking intuitively in terms of hashing, from an indexing perspective a rewrite system with rules having the same left-hand sides is as bad as a hash table in which all accesses are collisions.

Ideally, in an efficient implementation of RSEC one would like to adapt/modify indexing techniques, which currently work for context-insensitive term rewriting, or to invent new techniques serving the same purpose. This seems highly non-trivial and tedious, though. An alternative is to device embedding transformations of RSEC into rewriting logic that take better or full advantage of existing, context-insensitive indexing. Without context-sensitive indexing or other bookkeeping mechanisms hardwired in the reduction engine, due to the inherent non-determinism in parsing/splitting syntax into contextual representations, in the worst case one needs to search the entire term to find a legal position where a reduction can take place. While there does not seem that we can do much to avoid such an exhaustive search in the worst case, note that our first embedding in Figure 3.33 initiates such a search in the condition of every rewrite rule: since in practice many/most of the rewrite rules generated by the procedure in Figure 3.33 end up having the same left-hand side, the expensive search for appropriate splittings is potentially invoked many times. What we'd like to achieve is: (1) activate the expensive search for splitting only once; and (2) for each found split, quickly test which rule applies and apply it. Such a quick test as desired in (2) can be achieved for free on existing rewrite systems that use indexing, such as Maude, if one slightly modifies the embedding translation of RSEC into rewriting logic as shown in Figure 3.34.

The main idea is to keep the structure of the left-hand side of the RSEC rules in the left-hand side of the corresponding rewrite rules. This structure is crucial for indexing. To allow it, one needs to do the necessary splitting as a separate step. The first type of rewrite rules in Figure 3.34, one per term appearing as a left-hand side in any of the conditional rules generated following the first embedding in Figure 3.33, enables the splitting process on the corresponding contextual representations in the left-hand side of the original RSEC rule. We only define such rules for left-hand-side terms having at least one subterm in contextual representation, because if the left-hand side l has no such terms then the rule would be $\circ\bar{l} \to T$ **if** $\circ\bar{l} \to T$, which is useless and does not terminate.

The second type of rules in Figure 3.34, one per RSEC rule, have almost the same left-hand sides as the original RSEC rules; the only difference is the algebraic notation (as reflected by the overlining). Their right-hand sides plug the context representations, so that they always yield terms which are well-formed over the original syntax (possibly extended with auxiliary syntax for semantics components—configurations, states, etc.). Consider, for example, the six RSEC rules

rules:

// for each term l that appears as left-hand side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \to \ldots$ with $n > 0$, add the following
// conditional rewrite rule (there could be one l for many reduction rules):

$$\circ\, \overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \circ\, \overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n)) \to T$$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following (unconditional) semantic rewrite rule:

$$\circ\, \overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \to \overline{r}(\mathit{plug}(\overline{c'_1}[\overline{r_1}]), \ldots, \mathit{plug}(\overline{c'_n}[\overline{r_{n'}}]))$$

Figure 3.34: Second embedding of RSEC into rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{2}}_{\mathrm{RSEC}}$).

discussed in the preamble of Section 3.7, whose translation into rewrite rules following our first embedding in Figure 3.33 was discussed right above Theorem 7. Let us first consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top. Since in all these rules the contextual representation appears at the top of their left-hand side, which in terms of the first embedding in Figure 3.33 means that their corresponding rewrite rules (in the first embedding) had the form $\circ\, \mathit{Cfg} \to \ldots$ **if** $\mathit{split}(\mathit{Cfg}) \to \ldots$, we only need to add one rule of the first type in Figure 3.34 for them, namely (l is the identity pattern, i.e., $\overline{l}$ is a variable):

$$\circ\, \mathit{Cfg} \to \mathit{Cfg}' \;\; \textbf{if} \;\; \circ\, \mathit{split}(\mathit{Cfg}) \to \mathit{Cfg}'$$

With this, the six rewrite rules of the second type in Figure 3.34 corresponding to the six RSEC rules under discussion are the following:

$$\begin{aligned}
\circ\, C[I_1 \texttt{ <= } I_2] &\to \mathit{plug}(C[I_1 \leq_{\mathit{Int}} I_2]) \\
\circ\, C[\texttt{ skip ; } S_2] &\to \mathit{plug}(C[S_2]) \\
\circ\, C[\texttt{ if true then } S_1 \texttt{ else } S_2] &\to \mathit{plug}(C[S_1]) \\
\circ\, C[\texttt{ if false then } S_1 \texttt{ else } S_2] &\to \mathit{plug}(C[S_2]) \\
\circ\, \langle C, \sigma \rangle[X] &\to \mathit{plug}(\langle C, \sigma \rangle[\sigma(X)]) \;\; \textbf{if} \;\; \sigma(X) \neq \bot \\
\circ\, \langle C, \sigma \rangle[X \texttt{ := } I] &\to \mathit{plug}(\langle C, \sigma[I/X] \rangle[\texttt{ skip }]) \;\; \textbf{if} \;\; \sigma(X) \neq \bot
\end{aligned}$$

If one prefers the second variant for the reduction rules of lookup and assignment, namely

$$\begin{aligned}
\langle c[x], \sigma \rangle &\to \langle c[\sigma(x)], \sigma \rangle \quad \text{if } \sigma(x) \neq \bot \\
\langle c[x \texttt{ := } i], \sigma \rangle &\to \langle c[\texttt{ skip }], \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

then, since the left-hand side of these rules is a pattern of the form $\langle \mathit{Stmt}, \sigma \rangle$ which in algebraic form (over-lined) becomes a term of the form $\langle S, \sigma \rangle$, we need to add one more rewrite rule of the first type in Figure 3.34, namely

$$\circ\, \langle S, \sigma \rangle \to \mathit{Cfg}' \;\; \textbf{if} \;\; \circ\, \langle \mathit{split}(S), \sigma \rangle \to \mathit{Cfg}',$$

and to replace the rewrite rules for lookup and assignment above with the following two rules:

$$\begin{aligned}
\circ\, \langle C[X], \sigma \rangle &\to \langle \mathit{plug}(C[\sigma(X)]), \sigma \rangle \;\; \textbf{if} \;\; \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma \rangle &\to \langle \mathit{plug}(C[\texttt{ skip }]), \sigma[I/X] \rangle \;\; \textbf{if} \;\; \sigma(X) \neq \bot
\end{aligned}$$

Theorem 8. ***(Second faithful embedding of reduction semantics in rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.34. Then*

1. ***(step-for-step correspondence)*** $\text{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \circ\,\overline{t} \to^{1} \overline{t'}$ *using the corresponding rewrite rules obtained like in Figure 3.34 (first a conditional rule of the first type whose left-hand side matches* t*, then a rule of the second type which solves, in one rewrite step, the condition of the first rule); moreover, the reduction rule and the corresponding rewrite rules apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 6);*

2. ***(computational correspondence)*** $\text{RSEC} \vdash t \to^{\star} t'$ *iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \star\,\overline{t} \to \star\,\overline{t'}$.

Theorem 8 tells us that we can use our second rewriting logic embedding transformation in Figure 3.34 to seamlessly execute RSEC definitions on context-insensitive rewrite engines, such as Maude. This was also the case for our first embedding (Figure 3.33 and its corresponding Theorem 7). However, as explained above, in our second embedding the left-hand-side terms of the rewrite rules corresponding to the actual reduction semantics rules (the second type of rule in Figure 3.34) preserve the structure of the left-hand-side terms of the original corresponding reduction rules. This important fact has two benefits. On the one hand, the underlying rewrite engines can use that structure to enhance the efficiency of rewriting by means of indexing, as already discussed above. On the other hand, the resulting rewrite rules resemble the original reduction rules, so the language designer who wants to use our embedding feels more comfortable. Indeed, since the algebraic representation of terms (the overline) should not change the way they are perceived by a user, the only difference between the left-hand side of the original reduction rule and the left-hand side of the resulting rewrite rule is the $\circ$ symbol: $l(c_1[l_1], \ldots, c_n[l_n])$ versus $\circ\,\overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}])$, e.g., $\langle c[x := i], \sigma\rangle$ versus $\circ\,\langle C[X := I], \sigma\rangle$, where c, σ, x, i are reduction rule parameters while C, σ, X, I are corresponding variables of appropriate sorts.

Even though the representational distance between the left-hand-side terms in the original reduction rules and the left-hand-side terms in the resulting rewrite rules is minimal (one cannot eliminate the $\circ$, as extensively discussed in Section 3.3.3), unfortunately, the same does not hold true for the right-hand-side terms. Indeed, a right-hand side $r(c'_1[r_1], \ldots, c'_n[r_{n'}])$ of a reduction rule becomes the right-hand side $\overline{r}(\mathit{plug}(\overline{c'_1}[\overline{r_1}]), \ldots, \mathit{plug}(\overline{c'_n}[\overline{r_{n'}}]))$ of its corresponding rewrite rule, e.g., $\langle c[\,\texttt{skip}\,], \sigma\rangle$ becomes $\langle \mathit{plug}(C[\,\texttt{skip}\,]), \sigma\rangle$.

Figure 3.35 shows our third and final embedding of RSEC in rewriting logic, which has the advantage that it completely isolates the uses of *split*/*plug* from the semantic rewrite rules. Indeed, the rewrite rule associated to a reduction rule has the same left-hand side as in the second embedding, but now the right-hand side is actually the algebraic variant of the right-hand side of the original reduction rule. This is possible because of two simple adjustments of the second embedding:

1. To avoid having to explicitly use the *plug* operation in the semantic rewrite rules, we replace the first type of conditional rewrite rules in the second embedding, namely

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \circ\,\overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n)) \to T,$$

   with slightly modified conditional rewrite rules of the form

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \mathit{plug}(\circ\,\overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n))) \to T.$$

rules:

// for each term l that appears as the left-hand side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \to \ldots$, add the following conditional
// rewrite rule (there could be one l for many reduction rules):

$\circ\,\overline{l}(T_1, \ldots, T_n) \to T$ **if** $plug(\circ\,\overline{l}(split(T_1), \ldots, split(T_n))) \to T$

// for each non-identity term r appearing as right-hand side in a reduction rule
// $\ldots \to r(c_1[r_1], \ldots, c_n[r_n])$, add the following equation
// (there could be one r for many reduction rules):

$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n))$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following semantic rewrite rule:

$\circ\,\overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \to \overline{r}(\overline{c'_1}[\overline{r_1}], \ldots, \overline{c'_n}[\overline{r_{n'}}])$

Figure 3.35: Third embedding of RSEC in rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}}$).

Therefore, the left-hand-side term of the condition is wrapped with the *plug* operation. Since rewriting is context-insensitive, the *plug* wrapper does not affect the rewrites that happen underneath in the $\circ\,\overline{l}(\ldots)$ term. Like in the second embedding, the only way for $\circ$ to disappear from the condition left-hand side is for a semantic rule to apply. When that happens, the left-hand side of the condition is rewritten to a term of the form $plug(t)$, where t matches the right-hand side of some reduction semantics rule, which may potentially contain some subterms in contextual representation.

2. To automatically plug all the subterms in contextual representation that appear in t after the left-hand-side term of the condition in the rule above rewrites to $plug(t)$, we add equations of the form

$$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n)),$$

one for each non-identity pattern r appearing as a right-hand side of an RSEC rule; if r is an identity pattern then the equation becomes $plug(Syn) = plug(Syn)$, so we omit it.

Let us exemplify our third rewriting logic embedding transformation of reduction semantics with evaluation contexts using the same six reduction rules used so far in this section, but, to make it more interesting, considering the second variant of reduction rules for variable lookup and assignment. We have two left-hand-side patterns in these reduction rules, namely *Configuration* and $\langle Stmt, \sigma \rangle$, so we have the following two rules of the first type in Figure 3.35:

$$\begin{array}{c} \circ\, Cfg \to Cfg' \ \textbf{if}\ plug(\circ\, split(Cfg)) \to Cfg' \\ \circ\, \langle S, \sigma \rangle \to Cfg' \ \textbf{if}\ plug(\circ\, \langle split(S), \sigma \rangle) \to Cfg' \end{array}$$

We also have two right-hand-side patterns in these reduction rules, the same two as above, but the

first one is an identity pattern so we only add one equation of the second type in Figure 3.35:

$$plug(\langle C[Syn], \sigma\rangle) = \langle plug(C[Syn]), \sigma\rangle$$

We can now give the six rewrite rules corresponding to the six reduction rules in discussion:

$$\begin{aligned}
\circ\, C[I_1 \texttt{ <= } I_2] &\to C[I_1 \leq_{Int} I_2] \\
\circ\, C[\texttt{ skip ; } S_2] &\to C[S_2] \\
\circ\, C[\texttt{ if true then } S_1 \texttt{ else } S_2] &\to C[S_1] \\
\circ\, C[\texttt{ if false then } S_1 \texttt{ else } S_2] &\to C[S_2] \\
\circ\, \langle C[X], \sigma\rangle &\to \langle C[\sigma(X)], \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma\rangle &\to \langle C[\texttt{ skip }], \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot
\end{aligned}$$

The six rewrite rules above are as close to the original reduction semantics rules as one can hope them to be in a rewriting setting. Note that, for simplicity, we preferred to desugar the characteristic rule of reduction semantics with evaluation contexts in all our examples in this subsection. At this moment we have all the infrastructure needed to also include a rewrite equivalent of it:

$$\circ\, C[Syn] \to C[Syn'] \ \textbf{if}\ C \neq \Box \wedge \circ\, Syn \to Syn'$$

Note that we first check whether the context is proper in the condition of the characteristic rewrite rule above, and then we initiate a (small-step) reduction of the redex (by tagging it with the symbol $\circ$). The condition is well-defined in rewriting logic because, as explained in Figure 3.32, we subsorted all the syntactic sorts together with the configuration under the top sort *Syntax*, so all these sorts belong to the same kind (see Section 2.7), which means that the operation $\circ$ can apply to any of them, including to *Syntax*, despite the fact that it was declared to take a *Configuration* to an *ExtendedConfiguration* (like in Section 3.3.3). With this characteristic rewrite rule, we can now restate the six rewrite rules corresponding to the six reduction rules above as follows:

$$\begin{aligned}
\circ\, I_1 \texttt{ <= } I_2 &\to I_1 \leq_{Int} I_2 \\
\circ\, \texttt{ skip ; } S_2 &\to S_2 \\
\circ\, \texttt{ if true then } S_1 \texttt{ else } S_2 &\to S_1 \\
\circ\, \texttt{ if false then } S_1 \texttt{ else } S_2 &\to S_2 \\
\circ\, \langle C[X], \sigma\rangle &\to \langle C[\sigma(X)], \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma\rangle &\to \langle C[\texttt{ skip }], \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot
\end{aligned}$$

Note, again, that $\circ$ is applied on arguments of various sorts in the same kind with *Configuration*.

The need for $\circ$ in the left-hand-side terms of rules like above is now even more imperative than before. In addition to all the reasons discussed so far, there are additional reasons now for which the dropping of $\circ$ would depart us from the intended faithful capturing of reduction semantics in rewriting logic. Indeed, if we drop $\circ$ then there is nothing to stop the applications of rewrite rules at any places in the term to rewrite, potentially including places which are not allowed to be evaluated yet, such as, for example, in the branches of a conditional. Moreover, such applications of rules could happen concurrently, which is strictly disallowed by reduction semantics with or without evaluation contexts. The role of $\circ$ is precisely to inhibit the otherwise unrestricted potential to apply rewrite rules everywhere and concurrently: rules are now applied sequentially and only at the top of the original term, exactly like in reduction semantics.

sorts:
$\mathit{Configuration}$, $\mathit{ExtendedConfiguration}$
subsort:
$\mathit{Configuration} < \mathit{ExtendedConfiguration}$
operations:
$\langle \_, \_ \rangle$: $\mathit{Stmt} \times \mathit{State} \to \mathit{Configuration}$
$\langle \_ \rangle$: $\mathit{Pgm} \to \mathit{Configuration}$
$\circ \_$: $\mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce one step
$\star \_$: $\mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce all steps
rule:
$\star\, \mathit{Cfg} \to \star\, \mathit{Cfg}'$ **if** $\circ\, \mathit{Cfg} \to \mathit{Cfg}'$ // where $\mathit{Cfg}, \mathit{Cfg}'$ are variables of sort $\mathit{Configuration}$

Figure 3.36: Configurations and infrastructure for the rewriting logic embedding of RSEC(IMP).

Theorem 9. ***(Third faithful embedding of reduction semantics into rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition (with or without a characteristic reduction rule) and let* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.35 (plus the characteristic rewrite rule above in case* RSEC *comes with a characteristic reduction rule). Then*

1. ***(step-for-step correspondence)*** $\mathrm{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}} \vdash \circ\, \overline{t} \to^1 \overline{t'}$*;*

2. ***(computational correspondence)*** $\mathrm{RSEC} \vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}} \vdash \star\, \overline{t} \to \star\, \overline{t'}$*.*

We can therefore safely conclude that RSEC has been captured as a methodological fragment of rewriting logic. The faithful embeddings of reduction semantics into rewriting logic above can be used in at least two different ways. On the one hand, they can be used as compilation steps transforming a context-sensitive reduction system into an equivalent context-insensitive rewrite system, which can be further executed/compiled/analyzed using conventional rewrite techniques and existing rewrite engines. On the other hand, the embeddings above are so simple, that one can simply use them manually and thus "think reduction semantics" in rewriting logic.

Reduction Semantics with Evaluation Contexts of IMP in Rewriting Logic

We here discuss the complete reduction semantics with evaluation contexts definition of IMP in rewriting logic, obtained by applying the faithful embedding techniques discussed above to the reduction semantics definition of IMP in Figure 3.31 in Section 3.7.1. We start by defining the needed configurations, then we give all the rewrite rules and equations embedding the evaluation contexts and their splitting/plugging mechanism in rewriting logic, and then we finally give three rewrite theories corresponding to the three embeddings discussed above, each including the (same) configurations definition and embedding of evaluation contexts.

Figure 3.36 gives an algebraic definition of IMP configurations as needed for reduction semantics with evaluation contexts, together with the additional infrastructure needed to represent the one-step and multi-step transition relations. Everything defined in Figure 3.36 has already been discussed

in the context of small-step SOS (see Figures 3.13 and 3.17 in Section 3.3.3). Note, however, that we only defined a subset of the configurations needed for small-step SOS, more precisely only the top-level configurations (ones holding a program and ones holding a statement and a state). The intermediate configurations holding expressions and a state in small-step SOS are not needed here because reduction semantics with evaluation contexts does not need to explicitly decompose bigger reduction tasks into smaller ones until a redex is eventually found, like small-step SOS does; instead, the redex is found atomically by splitting the top level configuration into a context and the redex.

Figure 3.37 shows the rewriting logic theory $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ associated to the evaluation contexts of IMP in RSEC(IMP) (Figure 3.30) following the procedure described in Section 3.7.3 and summarized in Figure 3.32. Recall that all language syntactic categories and configurations are sunk into a top sort *Syntax*, and that one rule for splitting and one equation for plugging are generated for each context production. In general, the embedding of evaluation contexts tends to be the largest and the most boring portion of the rewriting logic embedding of a reduction semantics language definition. However, fortunately, this can be generated fully automatically. An implementation of the rewriting logic embedding techniques discussed in this section may even completely hide this portion from the user. We show it in Figure 3.37 only for the sake of completeness.

Figure 3.38 shows the rewriting logic theory $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ corresponding to the rules in the reduction semantics with evaluation contexts of IMP in Section 3.7.1, following our first embedding transformation depicted in Figure 3.33. Like before, we used the rewriting logic convention that variables start with upper-case letters; if they are Greek letters, then we use a similar but larger symbol (e.g., Σ instead of σ for variables of sort *State*). These rules are added, of course, to those corresponding to evaluation contexts in Figure 3.37 (which are common to all three embeddings). Note that there is precisely one conditional rewrite rule in Figure 3.38 corresponding to each reduction semantics rule of IMP in Figure 3.31. Also, note that if a rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3. For example, the last reduction rule in Figure 3.31 results in the last rewrite rule in Figure 3.38, which is identical to the last rewrite rule corresponding to the small-step SOS of IMP in Figure 3.18. The rules that make use of evaluation contexts perform explicit splitting (in the left-hand side of the condition) and plugging (in the right-hand side of the conclusion) operations. As already discussed but worth reemphasizing, the main drawbacks of this type of rewriting logic embedding are: (1) the expensive, non-deterministic search involving splitting of the original term is performed for any rule, and (2) it does not take advantage of one of the major optimizations of rewrite engines, indexing, which allows for quick detection of matching rules based on the structure of their left-hand-side terms.

Figure 3.39 shows the rewriting logic theory $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ that follows our second embedding transformation depicted in Figure 3.34. These rules are also added to those corresponding to evaluation contexts in Figure 3.37. Note that now there is precisely one *unconditional* rewrite rule corresponding to each reduction semantics rule of IMP in Figure 3.31 and that, unlike in the first embedding in Figure 3.38, the left-hand side of each rule preserves the exact structure of the left-hand side of the original reduction rule (after desugaring of the characteristic rule), so this embedding takes advantage of indexing optimizations in rewrite engines. Like in the first embedding, if a reduction rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3 (e.g., the last rule). Unlike in the first embedding, we also need to add a generic conditional rule, the first one in Figure 3.39, which initiates the splitting. We need only one rule of this type because

sorts:

$Syntax,\ Context$

subsorts:

$AExp, BExp, Stmt, Configuration < Syntax$

operations:

$\square : \to Context$ $\qquad$ $\_[\_] : Context \times Syntax \to Syntax$

$split : Syntax \to Syntax$ $\qquad$ $plug : Syntax \to Syntax$

$\langle \_, \_ \rangle : Context \times State \to Context$

$\_ + \_ : Context \times AExp \to Context$ $\qquad$ $\_ + \_ : AExp \times Context \to Context$

$\_ / \_ : Context \times AExp \to Context$ $\qquad$ $\_ / \_ : AExp \times Context \to Context$

$\_ <= \_ : Context \times AExp \to Context$ $\qquad$ $\_ <= \_ : Int \times Context \to Context$

$\_ \texttt{and} \_ : Context \times BExp \to Context$

$\texttt{not} \_ : Context \to Context$

$\_ := \_ : Id \times Context \to Context$

$\_ ; \_ : Context \times Stmt \to Context$

$\texttt{if} \_ \texttt{then} \_ \texttt{else} \_ : Context \times Stmt \times Stmt \to Context$

rules and equations:

$split(Syn) \to \square[Syn]$ $\qquad$ $plug(\square[Syn]) = Syn$

$split(\langle S, \sigma \rangle) \to \langle C, \sigma \rangle[Syn]$ **if** $split(S) \to C[Syn]$

$plug(\langle C, \sigma \rangle[Syn]) = \langle plug(C[Syn]), \sigma \rangle$

$split(A_1 + A_2) \to (C + A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C + A_2)[Syn]) = plug(C[Syn]) + A_2$

$split(A_1 + A_2) \to (A_1 + C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((A_1 + C)[Syn]) = A_1 + plug(C[Syn])$

$split(A_1 / A_2) \to (C / A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C / A_2)[Syn]) = plug(C[Syn]) / A_2$

$split(A_1 / A_2) \to (A_1 / C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((A_1 / C)[Syn]) = A_1 / plug(C[Syn])$

$split(A_1 <= A_2) \to (C <= A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C <= A_2)[Syn]) = plug(C[Syn]) <= A_2$

$split(I_1 <= A_2) \to (I_1 <= C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((I_1 <= C)[Syn]) = I_1 <= plug(C[Syn])$

$split(\texttt{not}\ B) \to (\texttt{not}\ C)[Syn]$ **if** $split(B) \to C[Syn]$

$plug((\texttt{not}\ C)[Syn]) = \texttt{not}\ plug(C[Syn])$

$split(B_1\ \texttt{and}\ B_2) \to (C\ \texttt{and}\ B_2)[Syn]$ **if** $split(B_1) \to C[Syn]$

$plug((C\ \texttt{and}\ B_2)[Syn]) = plug(C[Syn])\ \texttt{and}\ B_2$

$split(X := A) \to (X := C)[Syn]$ **if** $split(A) \to C[Syn]$

$plug((X := C)[Syn]) = X := plug(C[Syn])$

$split(S_1\ ;\ S_2) \to (C\ ;\ S_2)[Syn]$ **if** $split(S_1) \to C[Syn]$

$plug((C\ ;\ S_2)[Syn]) = plug(C[Syn])\ ;\ S_2$

$split(\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \to (\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2)[Syn]$ **if** $split(B) \to C[Syn]$

$plug((\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2)[Syn]) = \texttt{if}\ plug(C[Syn])\ \texttt{then}\ S_1\ \texttt{else}\ S_2$

Figure 3.37: $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$: Rewriting logic embedding of IMP evaluation contexts. The implicit split/plug reduction semantics mechanism is replaced by explicit rewriting logic sentences.

$\circ\ Cfg \rightarrow plug(\langle C, \sigma \rangle[\sigma(X)])$ **if** $split(Cfg) \rightarrow \langle C, \sigma \rangle[X] \ \wedge\ \sigma(X) \neq \bot$
$\circ\ Cfg \rightarrow plug(C[I_1 +_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 \texttt{ + } I_2]$
$\circ\ Cfg \rightarrow plug(C[I_1 /_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 \texttt{ / } I_2] \ \wedge\ I_2 \neq 0$
$\circ\ Cfg \rightarrow plug(C[I_1 \leq_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 \texttt{ <= } I_2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{ false }])$ **if** $split(Cfg) \rightarrow C[\texttt{ not true }]$
$\circ\ Cfg \rightarrow plug(C[\texttt{ true }])$ **if** $split(Cfg) \rightarrow C[\texttt{ not false }]$
$\circ\ Cfg \rightarrow plug(C[B_2])$ **if** $split(Cfg) \rightarrow C[\texttt{ true and } B_2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{ false }])$ **if** $split(Cfg) \rightarrow C[\texttt{ false and } B_2]$
$\circ\ Cfg \rightarrow plug(\langle C, \sigma[I/X] \rangle[\texttt{ skip }])$ **if** $split(Cfg) \rightarrow \langle C, \sigma \rangle[X \texttt{ := } I] \ \wedge\ \sigma(X) \neq \bot$
$\circ\ Cfg \rightarrow plug(C[S_2])$ **if** $split(Cfg) \rightarrow C[\texttt{ skip ; } S_2]$
$\circ\ Cfg \rightarrow plug(C[S_1])$ **if** $split(Cfg) \rightarrow C[\texttt{ if true then } S_1 \texttt{ else } S_2]$
$\circ\ Cfg \rightarrow plug(C[S_2])$ **if** $split(Cfg) \rightarrow C[\texttt{ if false then } S_1 \texttt{ else } S_2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{ if } B \texttt{ then } (S \texttt{ ; while } B \texttt{ do } S) \texttt{ else skip }])$ **if** $split(Cfg) \rightarrow C[\texttt{ while } B \texttt{ do } S]$
$\circ \langle \texttt{ var } Xl \texttt{ ; } S \rangle \rightarrow \langle S, (Xl \mapsto 0) \rangle$

Figure 3.38: $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the first embedding of the reduction semantics with evaluation contexts of IMP.

$$\circ\ Cfg \rightarrow Cfg' \ \textbf{if}\ \circ\ split(Cfg) \rightarrow Cfg'$$

$$\begin{aligned}
\circ \langle C, \sigma \rangle[X] &\rightarrow plug(\langle C, \sigma \rangle[\sigma(X)]) \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ C[I_1 \texttt{ + } I_2] &\rightarrow plug(C[I_1 +_{Int} I_2]) \\
\circ C[I_1 \texttt{ / } I_2] &\rightarrow plug(C[I_1 /_{Int} I_2]) \ \textbf{if}\ I_2 \neq 0 \\
\circ C[I_1 \texttt{ <= } I_2] &\rightarrow plug(C[I_1 \leq_{Int} I_2]) \\
\circ C[\texttt{ not true }] &\rightarrow plug(C[\texttt{ false }]) \\
\circ C[\texttt{ not false }] &\rightarrow plug(C[\texttt{ true }]) \\
\circ C[\texttt{ true and } B_2] &\rightarrow plug(C[B_2]) \\
\circ C[\texttt{ false and } B_2] &\rightarrow plug(C[\texttt{ false }]) \\
\circ \langle C, \sigma \rangle[X \texttt{ := } I] &\rightarrow plug(\langle C, \sigma[I/X] \rangle[\texttt{ skip }]) \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ C[\texttt{ skip ; } S_2] &\rightarrow plug(C[S_2]) \\
\circ C[\texttt{ if true then } S_1 \texttt{ else } S_2] &\rightarrow plug(C[S_1]) \\
\circ C[\texttt{ if false then } S_1 \texttt{ else } S_2] &\rightarrow plug(C[S_2]) \\
\circ C[\texttt{ while } B \texttt{ do } S] &\rightarrow plug(C[\texttt{ if } B \texttt{ then } (S \texttt{ ; while } B \texttt{ do } S) \texttt{ else skip }]) \\
\circ \langle \texttt{ var } Xl \texttt{ ; } S \rangle &\rightarrow \langle S, (Xl \mapsto 0) \rangle
\end{aligned}$$

Figure 3.39: $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the second embedding of the reduction semantics with evaluation contexts of IMP.

$$
\begin{aligned}
\circ\, Cfg &\to Cfg' \ \textbf{if}\ \circ\, plug(split(Cfg)) \to Cfg' \\
\circ\, C[Syn] &\to C[Syn'] \ \textbf{if}\ C \neq \Box \wedge \circ\, Syn \to Syn' \\
\\
\circ\, \langle C, \sigma\rangle[X] &\to \langle C, \sigma\rangle[\sigma(X)] \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, I_1 \,\texttt{+}\, I_2 &\to I_1 +_{Int} I_2 \\
\circ\, I_1 \,\texttt{/}\, I_2 &\to I_1 /_{Int} I_2] \ \textbf{if}\ I_2 \neq 0 \\
\circ\, I_1 \,\texttt{<=}\, I_2 &\to I_1 \leq_{Int} I_2 \\
\circ\, \texttt{not true} &\to \texttt{false} \\
\circ\, \texttt{not false} &\to \texttt{true} \\
\circ\, \texttt{true and}\, B_2 &\to B_2 \\
\circ\, \texttt{false and}\, B_2 &\to \texttt{false} \\
\circ\, \langle C, \sigma\rangle[X := I] &\to \langle C, \sigma[I/X]\rangle[\,\texttt{skip}\,] \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \texttt{skip ;}\, S_2 &\to S_2 \\
\circ\, \texttt{if true then}\, S_1 \,\texttt{else}\, S_2 &\to S_1 \\
\circ\, \texttt{if false then}\, S_1 \,\texttt{else}\, S_2 &\to S_2 \\
\circ\, \texttt{while}\, B \,\texttt{do}\, S &\to \texttt{if}\, B \,\texttt{then}\, (S \,\texttt{;}\, \texttt{while}\, B \,\texttt{do}\, S) \,\texttt{else skip} \\
\circ\, \langle\, \texttt{var}\, Xl \,\texttt{;}\, S\rangle &\to \langle S, (Xl \mapsto 0)\rangle
\end{aligned}
$$

Figure 3.40: $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the third embedding of the reduction semantics with evaluation contexts of IMP.

all the left-hand-side terms of reduction rules of IMP in Figure 3.31 that contain a subterm in contextual representation contain that term at the top. As already discussed, if one preferred to write, e.g., the lookup RSEC rule as $\langle c[x], \sigma\rangle \to \langle c[\sigma(x)], \sigma\rangle$ if $\sigma(x) \neq \bot$, then one would need an additional generic rule, namely $\circ\, \langle S, \sigma\rangle \to Cfg'$ **if** $\circ\, \langle split(S), \sigma\rangle \to Cfg'$. While these generic rules take care of splitting and can be generated relatively automatically, the remaining rewrite rules that correspond to the reduction rules still make explicit use of the internal (to the embedding) *plug* operation, which can arguably be perceived by language designers as an inconvenience.

Figure 3.40 shows the rewriting logic theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ obtained by applying our third embedding (shown in Figure 3.35). These rules are also added to those corresponding to evaluation contexts in Figure 3.37 and, like in the second embedding, there is precisely one unconditional rewrite rule corresponding to each RSEC rule of IMP. We also need to add a generic conditional rule, the first one, which completely encapsulates the rewriting logic representation of the splitting/plugging mechanism, so that the language designer can next focus exclusively on the semantic rules rather than on their representation in rewriting logic. The second rewrite rule in Figure 3.40 corresponds to the characteristic rule of reduction semantics with evaluation contexts and, as discussed, it is optional; if one includes it, as we did, we think that its definition in Figure 3.40 is as simple and natural as it can be. In what regards the remaining rewrite rules, the only perceivable difference between them and their corresponding reduction rules is that they are preceded by $\circ$.

All the above suggest that, in spite of its apparently advanced context-sensitivity and splitting/plugging mechanism, reduction semantics with evaluation contexts can be safely regarded as a methodological fragment of rewriting logic. Or, put differently, while context-sensitive reduction seems crucial for programming language semantics, it is in fact unnecessary. A conditional rewrite framework can methodologically achieve the same results, and as discussed in this chapter, so can do for the other conventional language semantics approaches.

The following corollary of Theorems 7, 8, and 9 establishes the faithfulness of the representations of the reduction semantics with evaluation contexts of IMP in rewriting logic:

Corollary 5. *For any* IMP *configurations* C *and* C'*, the following equivalences hold:*

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \end{aligned}$$

and

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to^\star C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \end{aligned}$$

Therefore, there is no perceivable computational difference between the reduction semantics with evaluation contexts RSEC(IMP) and its corresponding rewriting logic theories.

☆ Reduction Semantics with Evaluation Contexts of IMP in Maude

Figure 3.41 shows a Maude representation of the rewrite theory $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ in Figure 3.37 that embeds IMP's evaluation contexts by making explicit the split/plug mechanism which is implicit in reduction semantics with evaluation contexts. Figure 3.41 also includes the Maude definition of configurations (see Figure 3.36). We took the freedom to implement a simple optimization which works well in Maude, but which may not work as well in other engines or systems (which is why we did not incorporate it as part of the general procedure to represent reduction semantics with evaluation contexts in rewriting logic): we defined the contextual representation operation `_[_]` to have as result the *kind* (see Section 2.7) `[Syntax]` instead of the sort `Syntax`. This allows us to include the equation `plug(Syn) = Syn`, where `Syn` is a variable of *sort* `Syntax`, which gives us the possibility to also use terms which do not make use of contexts in the right-hand sides of rewrite rules. To test the rules for splitting, one can write Maude commands such as the one below, asking Maude to search for all splits of a given term:

```
search split(3 <= (2 + X) / 7) =>! Syn:[Syntax] .
```

The `!` tag on the arrow `=>` in the command above tells Maude to only report the normal forms, in this case the completed splits. As expected, Maude finds all seven splits and outputs the following:

```
Solution 1 (state 1)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> [][3 <= (2 + X) / 7]

Solution 2 (state 2)
states: 8  rewrites: 19 in ... cpu (. real) (0 rewrites/second)
Syn --> ([] <= (2 + X) / 7)[3]

Solution 3 (state 3)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= [])[(2 + X) / 7]

Solution 4 (state 4)
```

```
mod IMP-CONFIGURATIONS-EVALUATION-CONTEXTS is including IMP-SYNTAX + STATE .
  sorts Configuration ExtendedConfiguration .
  subsort Configuration < ExtendedConfiguration .
  op <_,_> : Stmt State -> Configuration .
  op <_> : Pgm -> Configuration .
  ops (o_) (*_) : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SPLIT-PLUG-EVALUATION-CONTEXTS is including IMP-CONFIGURATIONS-EVALUATION-CONTEXTS .
  sorts Syntax Context .  subsorts AExp BExp Stmt Configuration < Syntax .
  op [] : -> Context .  op _[_] : Context Syntax -> [Syntax] [prec 1] .
  ops split plug : Syntax -> Syntax . --- to split Syntax into context[redex]

  var X : Id .  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .
  var Sigma : State .  var I1 : Int .  var Syn : Syntax .  var C : Context .

  rl split(Syn) => [][Syn] .  eq plug([][Syn]) = Syn .  eq plug(Syn) = Syn .

  op <_,_> : Context State -> Context .  eq plug(< C, Sigma > [Syn]) = < plug(C[Syn]), Sigma > .
 crl split(< S, Sigma >) => < C, Sigma > [Syn] if split(S) => C[Syn] .

  op _+_ : Context AExp -> Context .  eq plug((C + A2)[Syn]) = plug(C[Syn]) + A2 .
 crl split(A1 + A2) => (C + A2)[Syn] if split(A1) => C[Syn] .
  op _+_ : AExp Context -> Context .  eq plug((A1 + C)[Syn]) = A1 + plug(C[Syn]) .
 crl split(A1 + A2) => (A1 + C)[Syn] if split(A2) => C[Syn] .

  op _/_ : Context AExp -> Context .  eq plug((C / A2)[Syn]) = plug(C[Syn]) / A2 .
 crl split(A1 / A2) => (C / A2)[Syn] if split(A1) => C[Syn] .
  op _/_ : AExp Context -> Context .  eq plug((A1 / C)[Syn]) = A1 / plug(C[Syn]) .
 crl split(A1 / A2) => (A1 / C)[Syn] if split(A2) => C[Syn] .

  op _<=_ : Context AExp -> Context .  eq plug((C <= A2)[Syn]) = plug(C[Syn]) <= A2 .
 crl split(A1 <= A2) => (C <= A2)[Syn] if split(A1) => C[Syn] .
  op _<=_ : Int Context -> Context .  eq plug((I1 <= C)[Syn]) = I1 <= plug(C[Syn]) .
 crl split(I1 <= A2) => (I1 <= C)[Syn] if split(A2) => C[Syn] .

  op not_ : Context -> Context .  eq plug((not C)[Syn]) = not plug(C[Syn]) .
 crl split(not B) => (not C)[Syn] if split(B) => C[Syn] .

  op _and_ : Context BExp -> Context .  eq plug((C and B2)[Syn]) = plug(C[Syn]) and B2 .
 crl split(B1 and B2) => (C and B2)[Syn] if split(B1) => C[Syn] .

  op _:=_ : Id Context -> Context .  eq plug((X := C)[Syn]) = X := plug(C[Syn]) .
 crl split(X := A) => (X := C)[Syn] if split(A) => C[Syn] .

  op _;_ : Context Stmt -> Context .  eq plug((C ; S2)[Syn]) = plug(C[Syn]) ; S2 .
 crl split(S1 ; S2) => (C ; S2)[Syn] if split(S1) => C[Syn] .

  op if_then_else_ : Context Stmt Stmt -> Context .
 crl split(if B then S1 else S2) => (if C then S1 else S2)[Syn] if split(B) => C[Syn] .
  eq plug((if C then S1 else S2)[Syn]) = if plug(C[Syn]) then S1 else S2 .
endm
```

Figure 3.41: The configuration and evaluation contexts of IMP in Maude, as needed for the three variants of reduction semantics with evaluation contexts of IMP in Maude.

```
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ([] / 7))[2 + X]

Solution 5 (state 5)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= (([] + X) / 7))[2]

Solution 6 (state 6)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ((2 + []) / 7))[X]

Solution 7 (state 7)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= 2 + X / [])[7]
```

If, however, one replaces any of the rules for splitting with equations, then, as expected, one looses some of the splitting behaviors. For example, if one replaces the generic rule for splitting `rl split(Syn) => [][Syn]` by an apparently equivalent equation `eq split(Syn) = [][Syn]`, then Maude will be able to detect no other splitting of a term t except for $\Box[t]$ (because Maude executes the equations before the rules; see Section 2.8).

Figure 3.42 shows two Maude modules implementing the first two rewriting logic theories $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ (Figure 3.38) and $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ (Figure 3.39), and Figure 3.43 shows the Maude module implementing the third rewrite theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ (Figure 3.40), respectively. Each of these three Maude modules imports the module `IMP-CONFIGURATION-EVALUATION-CONTEXTS` defined in Figure 3.41 and is executable. Maude, through its rewriting capabilities, therefore yields an IMP reduction semantics with evaluation contexts interpreter for each of the three modules in Figures 3.42 and 3.43. For any of them, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 39356 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, n |-> 0 & s |-> 5050 >
state = (n |-> 0 , s |-> 5050) >
```

One can use any of the general-purpose tools provided by Maude on the reduction semantics with evaluation contexts definitions above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. Not unexpectedly, the same number of states as in the case of small-step SOS and MSOS will be discovered by this search command, namely 1509. Indeed, the splitting/cooling mechanism of RSEC is just another way to find where the next reduction step should take place; it does not generate any different reductions of the original configuration.

3.7.4 Notes

Reduction semantics with evaluation contexts was introduced by Felleisen and his collaborators (see, e.g., [29, 99]) as a variant small-step structural operational semantics. By making the evaluation

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg : Configuration .  var C : Context .

 crl o Cfg => plug(< C,Sigma >[Sigma(X)]) if split(Cfg) => < C,Sigma >[X]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[I1 +Int I2]) if split(Cfg) => C[I1 + I2] .
 crl o Cfg => plug(C[I1 /Int I2]) if split(Cfg) => C[I1 / I2]
  /\ I2 =/=Bool 0 .
 crl o Cfg => plug(C[I1 <=Int I2]) if split(Cfg) => C[I1 <= I2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[not true] .
 crl o Cfg => plug(C[true]) if split(Cfg) => C[not false] .
 crl o Cfg => plug(C[B2]) if split(Cfg) => C[true and B2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[false and B2] .
 crl o Cfg => plug(< C,Sigma[I / X] >[skip]) if split(Cfg) => < C,Sigma >[X := I]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[skip ; S2] .
 crl o Cfg => plug(C[S1]) if split(Cfg) => C[if true  then S1 else S2] .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[if false then S1 else S2] .
 crl o Cfg => plug(C[if B then (S ; while B do S) else skip]) if split(Cfg) => C[while B do S] .
  rl o < var Xl ; S > => < S,(Xl |-> 0) > .
endm



mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg Cfg' : Configuration .  var C : Context .

 crl o Cfg => Cfg' if o split(Cfg) => Cfg' .   --- generic rule enabling splitting

 crl o < C,Sigma >[X] => plug(< C,Sigma >[Sigma(X)])
  if Sigma(X) =/=Bool undefined .
  rl o C[I1 + I2] => plug(C[I1 +Int I2]) .
 crl o C[I1 / I2] => plug(C[I1 /Int I2])
  if I2 =/=Bool 0 .
  rl o C[I1 <= I2] => plug(C[I1 <=Int I2]) .
  rl o C[not  true] => plug(C[false]) .
  rl o C[not false] => plug(C[ true]) .
  rl o C[true  and B2] => plug(C[B2]) .
  rl o C[false and B2] => plug(C[false]) .
 crl o < C,Sigma >[X := I] => plug(< C,Sigma[I / X] >[skip])
  if Sigma(X) =/=Bool undefined .
  rl o C[skip ; S2] => plug(C[S2]) .
  rl o C[if  true then S1 else S2] => plug(C[S1]) .
  rl o C[if false then S1 else S2] => plug(C[S2]) .
  rl o C[while B do S] => plug(C[if B then (S ; while B do S) else skip]) .
  rl o < var Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.42: The first two reduction semantics with evaluation contexts of IMP in Maude.

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .  var Xl : List{Id} .
  var Sigma : State .  var Cfg Cfg' : Configuration .  var Syn Syn' : Syntax  .  var C : Context .

  crl o Cfg => Cfg' if plug(o split(Cfg)) => Cfg' .           --- generic rule enabling splitting
  crl o C[Syn] => C[Syn'] if C =/=Bool [] /\ o Syn => Syn' .  --- characteristic rule

 crl o < C,Sigma >[X] => < C,Sigma >[Sigma(X)]
  if Sigma(X) =/=Bool undefined .
  rl o I1 + I2 => I1 +Int I2 .
 crl o I1 / I2 => I1 /Int I2
  if I2 =/=Bool 0 .
  rl o I1 <= I2 => I1 <=Int I2 .
  rl o not true  => false .
  rl o not false => true  .
  rl o true  and B2 => B2 .
  rl o false and B2 => false .
 crl o < C,Sigma >[X := I] => < C,Sigma[I / X] >[skip]
  if Sigma(X) =/=Bool undefined .
  rl o skip ; S2 => S2 .
  rl o if  true then S1 else S2 => S1 .
  rl o if false then S1 else S2 => S2 .
  rl o while B do S => if B then (S ; while B do S) else skip .
  rl o < var Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.43: The third reduction semantics with evaluation contexts of IMP in Maude.

context explicit and modifiable, reduction semantics with evaluation contexts is considered by many to be a significant improvement over small-step SOS. Like small-step SOS, reduction semantics with evaluation contexts has been broadly used to give semantics to programming languages and to various calculi. We here only briefly mention some strictly related work.

How expensive is the splitting of a term into an evaluation context and a redex? Unfortunately, it cannot be more efficient than testing the membership of a word to a context-free grammar and the latter is expected to be cubic in the size of the original term (folklore). Indeed, consider G an arbitrary CFG whose start symbol is S and let G_C be the "evaluation context" CFG grammar adding a fresh "context" nonterminal C, a fresh terminal $\#$, and productions $C \to \Box \mid CS\#$. Then it is easy to see that a word α is in the language of G if and only if $\#\alpha\#$ can be split as a contextual representation (can only be $(\Box\alpha\#)[\#]$). Thus, we should expect, in the worst case, a *cubic* complexity to split a term into an evaluation context and a redex. An additional exponent needs to be added, thus making splitting expected to be a *quadratic* operation in the worst case, when nested contexts are allowed in rules (i.e., when the redex is itself a contextual representation). Unfortunately, this terrible complexity needs to be paid at each step of reduction, not to mention that the size of the program to reduce can also grow as it is reduced. One possibility to decrease this complexity is to attempt to incrementally compute at each step the evaluation context that is needed at the next step (like in refocusing; see below); however, in the worst case the right-hand sides of rules may contain no contexts, in which case a fresh split is necessary at each step.

Besides our own efforts, we are aware of three other attempts to develop executable engines for reduction semantics with evaluation contexts, which we discuss here in chronological order:

1. A specification language for syntactic theories with evaluation contexts is proposed by Xiao *et al.* [101, 100], together with a system which generates Ocaml interpreters from specifications. Although the compiler in [101, 100] is carefully engineered, as rightfully noticed by Danvy and Nielsen in [23] it cannot avoid the quadratic overhead due to the context-decomposition step. This is consistent with our own observations expressed at several places in this section, namely that the advanced parsing underlying reduction semantics with evaluation contexts is the most expensive part when one is concerned with execution. Fortunately, the splitting of syntax into context and redex can be and typically is taken for granted in theoretical developments, making abstraction of the complexity of its implementation.

2. A technique called *refocusing* is proposed by Danvy and Nielsen in [23, 22]. The idea underlying refocusing is to keep the program decomposed at all times (in a first-order continuation-like form) and to perform minimal changes to the resulting structure to find the next redex. Unfortunately, refocusing appears to work well only with restricted RSEC definitions, namely ones whose evaluation contexts grammar has the property of unique decomposition of a term into a context and a redex (so constructs like the non-deterministic addition of IMP are disallowed), and whose reduction rules are deterministic.

3. PLT-Redex, which is implemented in Scheme by Findler and his collaborators [43, 28], is perhaps the most advanced tool developed specifically to execute reduction semantics with evaluation contexts. PLT-Redex builds upon a direct implementation of context-sensitive reduction, so it cannot avoid the worst-case quadratic complexity of context decomposition, same as the interpreters generated by the system in [101, 100] discussed above. Several large language semantics engineering case studies using PLT-Redex are discussed in [28].

Our embeddings of reduction semantics with evaluation contexts into rewriting logic are inspired from a related embedding by Șerbănuță *et al.* in [85]. The embedding in [85] was similar to our third embedding here, but it included splitting rules also for terms in reducible form, e.g., $split(I_1 <= I_2) \rightarrow \Box[I_1 <= I_2]$. Instead, we preferred to include a generic rule $split(Syn) \rightarrow \Box[Syn]$ here, which allows us to more mechanically derive the rewrite rules for splitting from the CFG of evaluation contexts. Calculating the exact complexity of our approach seems to be hard, mainly because of optimizations employed by rewrite engines, e.g., indexing. Since at each step we still search for all the relevant splits of the term into an evaluation context and a redex, in the worst case we still pay the quadratic complexity. However, as suggested by the performance numbers in [85] comparing Maude running the resulting rewrite theory against PLT-Redex, which favor the former by a large margin, our embeddings may serve as alternative means to getting more efficient implementations of reduction semantics engines. There are strong reasons to believe that our third embedding can easily be automated in a way that the user never sees the split/plug operations.

3.7.5 Exercises

Exercise 108. *Suppose that one does not like mixing semantic components with syntactic evaluation contexts as we did above (by including the production* $Context ::= \langle Context, State \rangle$*). Instead, suppose that one prefers to work with configuration tuples like in SOS, holding the various components needed for the language semantics, the program or fragment of program being just one of them. In other words, suppose that one wants to make use of the contextual representation notation only on the*

syntactic component of configurations. In this case, the characteristic rule becomes

$$\frac{\langle e, \gamma\rangle \to \langle e', \gamma'\rangle}{\langle c[e], \gamma\rangle \to \langle c[e'], \gamma'\rangle}$$

where γ and γ' consist of configuration semantic components that are necessary to evaluate e and e', respectively, such as states, outputs, stacks, etc. Modify accordingly the six reduction semantics with evaluation contexts rules discussed at the beginning of Section 3.7.

The advantage of this approach is that it allows the evaluation contexts to be defined exclusively over the syntax of the language. However, configurations holding code and state still need to be defined. Moreover, many rules which looked compact before, such as i_1 `<=` $i_2 \to i_1 \leq_{Int} i_2$, will now look heavier, e.g., $\langle i_1$ `<=` $i_2, \sigma\rangle \to \langle i_1 \leq_{Int} i_2, \sigma\rangle$.

Exercise 109. *Like in Exercise 108, suppose that one does not like to mix syntactic and semantic components in evaluation contexts, but that, instead, one is willing to accept to slightly enrich the syntax of the programming language with a special statement construct Stmt ::=* `var` *Id* `=` *AExp which both declares and initializes a variable*[7]*. Then*

1. *Write structural identities that desugar the current top-level program variable declarations* `var` $x_1, ..., x_n$ `;` s *into statements of the form* `var` x_1 `=` 0 `;` ... `;` `var` x_n `=` 0 `;` s.
2. *Add a new context production that allows evaluation after the new variable declarations.*
3. *Modify the variable lookup and assignment rules discussed above so that one uses the new declarations instead of a state. Hint: the context should have the form* `var` x `=` i `;` c.

The advantage of this approach is that one does not need an explicit state anymore, so the resulting definition is purely syntactic. In fact, the state is there anyway, but encoded syntactically as a sequence of variable initializations preceding any other statement. This trick works in this case, but it cannot be used as a general principle to eliminate configurations in complex languages.

Exercise* 110. *Exercise 108 suggests that one can combine MSOS (Section 3.6) and evaluation contexts, in that one can use MSOS's labels to obtain modularity at the configuration level and one can use the evaluation contexts idea to detect and modify the contexts/redexes in the syntactic component of a configuration. Rewrite the six rules discussed at the beginning of Section 3.7 as they would appear in a hypothetical framework merging MSOS and evaluation contexts.*

Exercise 111. *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that* `/` *short-circuits when its numerator evaluates to* 0.
Hint: Make `/` *strict in only the first argument, then use a rule to reduce* 0`/`a_2 *to 0 and a rule to reduce* i_1`/`a_2 *to* i_1`/'`a_2 *when* $i_1 \neq 0$, *where* `/'` *is strict in its second argument, and finally a rule to reduce* i_1`/'`i_2 *to* $i_1 /_{Int} i_2$ *when* $i_2 \neq 0$.

Exercise 112. *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments.*

[7] Similar language constructs exist in many programming language (C, Java, etc.), so one may find their inclusion in the language acceptable.

Exercise 113. *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 108 (that is, use evaluation contexts only for the* IMP *language syntax, and handle the semantic components using configurations, like in SOS).*

Exercise 114. *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 109.*

Exercise* 115. *Give a semantics of* IMP *using the hypothetical framework combining reduction semantics with evaluation contexts and MSOS proposed in Exercise 110.*

Exercise 116. *Modify the rewrite theory* $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that later on one can define the reduction semantics of* / *to short-circuit when the numerator evaluates to* 0 *(as required in Exercises 118, 124, and 130).*

Exercise 117. *Modify the rewrite theory* $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that one can later on define the reduction semantics of conjunction to be non-deterministically strict in both its arguments (as required in Exercises 119, 125, and 131).*

Exercise 118. *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of* / *that short-circuits when the numerator evaluates to* 0 *(see also Exercise 116).*

Exercise 119. *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of conjunction that defines it as non-deterministically strict in both its arguments (see also Exercise 117).*

Exercise 120. *As discussed in several places so far in Section 3.7, the reduction semantics rules for variable lookup and assignment can also be given in a way in which their left-hand-side terms are not in contextual representation (i.e.,* $\langle c[x], \sigma\rangle$ *instead of* $\langle c, \sigma\rangle[x]$*, etc.). Modify the corresponding rewrite rules of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for this alternative reduction semantics.*

Exercise 121. *Modify the rewriting logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 113.*

Exercise 122. *Modify the rewriting logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 114.*

Exercise* 123. *Combining the underlying ideas of the embedding of MSOS in rewriting logic discussed in Section 3.6.3 and the embedding of reduction semantics with evaluation contexts in Figure 3.33, give a rewriting logic semantics of* IMP *corresponding to the semantics of* IMP *in Exercise 115.*

Exercise 124. *Same as Exercise 118, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 125. *Same as Exercise 119, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 126. *Same as Exercise 120, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 127. *Same as Exercise 121, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 128. *Same as Exercise 122, but for* $\mathcal{R}^{\boxed{2}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise* 129. *Same as Exercise 123, but for Figure 3.39 (instead of Figure 3.38).*

Exercise 130. *Same as Exercise 118, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise 131. *Same as Exercise 119, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise 132. *Same as Exercise 120, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise 133. *Same as Exercise 121, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise 134. *Same as Exercise 122, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

Exercise* 135. *Same as Exercise 123, but for Figure 3.40 (instead of Figure 3.38).*

Exercise 136. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that* / *short-circuits when its numerator evaluates to* 0 *(see also Exercises 111, 116, 118, 124, and 130).*

Exercise 137. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 112, 117, 119, 125, and 131).*

Exercise 138. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 120, 126, and 132.*

Exercise 139. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 121, 127, and 133.*

Exercise 140. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 122, 128, and 134.*

Exercise* 141. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the semantics in Exercises 123, 129, and 135.*

Exercise 142. *Same as Exercise 68, but for reduction semantics with evaluation contexts instead of small-step SOS: add variable increment to* IMP*, like in Section 3.7.2.*

Exercise 143. *Same as Exercise 72, but for reduction semantics with evaluation contexts instead of small-step SOS: add input/output to* IMP*, like in Section 3.7.2.*

Exercise* 144. *Consider the hypothetical framework combining MSOS with reduction semantics with evaluation contexts proposed in Exercise 110, and in particular the* IMP *semantics in such a framework in Exercise 115, its rewriting logic embeddings in Exercises 123, 129, and 135, and their Maude implementation in Exercise 141. Define the semantics of the input/output constructs above modularly first in the framework in discussion, then using the rewriting logic embeddings, and finally in Maude.*

Exercise 145. *Same as Exercise 77, but for reduction semantics with evaluation contexts instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.7.2.*

Exercise 146. *Same as Exercise 85, but for reduction semantics with evaluation contexts instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.7.2.*

Exercise 147. *Same as Exercise 90, but for reduction semantics with evaluation contexts instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.7.2.*

Exercise* 148. *This exercise asks to define* IMP++ *in reduction semantics, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the reduction semantics with evaluation contexts of* IMP++ *discussed in Section 3.7.2 instead of its small-step SOS in Section 3.5.6.*

3.8 The Chemical Abstract Machine (CHAM)

The *chemical abstract machine*, or the *CHAM*, is both a model of concurrency and a specific operational semantics style. The states of a CHAM are metaphorically regarded as chemical solutions formed with floating molecules. Molecules can interact with each other by means of reactions. A reaction can involve several molecules and can change them, delete them, and/or create new molecules. One of the most appealing aspects of the chemical abstract machine is that its reactions can take place concurrently, unrestricted by context. To facilitate local computation and to represent complex data-structures, molecules can be nested by encapsulating groups of molecules as sub-solutions. The chemical abstract machine was proposed as an alternative to SOS and its variants, including reduction semantics with evaluation contexts, in an attempt to circumvent their limitations, particularly their lack of support for true concurrency.

CHAM Syntax

The *basic molecules* of a CHAM are ordinary algebraic terms over a user-defined syntax. Several molecules wrapped within a membrane form a *solution*, which is also a molecule. The CHAM uses the symbols $\{\!|$ and $|\!\}$ to denote membranes. For example, $\{\!|m_1\ m_2\ \ldots\ m_k|\!\}$ is a solution formed with the molecules m_1, m_2, ..., m_k. The order of molecules in a solution is irrelevant, so a solution can be regarded as a multi-set, or a bag of molecules wrapped within a membrane. Since solutions are themselves molecules, we can have arbitrarily nested molecules. This nesting mechanism is generic for all CHAMs and is given by the following algebraic CFG:

$$\begin{array}{rcl} \mathit{Molecule} & ::= & \mathit{Solution} \mid \mathit{Molecule} \lhd \mathit{Solution} \\ \mathit{Solution} & ::= & \{\!|\mathbf{Bag\{}\mathit{Molecule}\mathbf{\}}|\!\} \end{array}$$

The operator $\_ \lhd \_$ is called the *airlock* operator and will be discussed shortly (under general CHAM laws), after we discuss the CHAM rules. When defining a CHAM, one is only allowed to extend the syntax of molecules, which implicitly also extends the syntax that the solution terms can use. However, one is not allowed to explicitly extend the syntax of solutions. In other words, solutions can only be built using the generic syntax above, on top of user-defined syntactic extensions of molecules. Even though we do not formalize it here (and we are not aware of other formulations elsewhere either), it is understood that one can have multiple types of molecules in a CHAM.

Specific CHAM Rules

In addition to extending the syntax of molecules, a CHAM typically also defines a set of *rules*, each rule being a rule schemata but called a rule for simplicity. A CHAM rule has the form

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$$

where $m_1, m_2, \ldots, m_k$ and $m'_1, m'_2, \ldots, m'_l$ are not necessarily distinct molecules (since CHAM rules are schemata, these molecule terms may contain meta-variables). Molecules appearing in a rule are restricted to contain only subsolution terms which are either solution meta-variables or otherwise have the form $\{\!|m|\!\}$, where m is some molecule term. For example, a CHAM rule cannot contain subsolution terms of the form $\{\!|m\ s|\!\}$, $\{\!|m_1\ m_2|\!\}$, or $\{\!|m_1\ m_2\ s|\!\}$, with m, m_1, m_2 molecule terms and s solution term, but it can contain ones of the form $\{\!|m|\!\}$, $\{\!|m_1 \lhd \{\!|m_2|\!\}|\!\}$, $\{\!|m \lhd s|\!\}$, etc. This restriction is justified by chemical intuitions, namely that matching inside a solution is a rather

complex operation which needs special handling (the airlock operator $\_ \lhd \_$ is used for this purpose). Note that CHAM rules are unconditional, that is, they have no premises.

General CHAM Laws

Any chemical abstract machine obeys the four laws below. Let CHAM be[8] a chemical abstract machine. Below we assume that mol, mol', mol_1, etc., are arbitrary concrete molecules of CHAM (i.e., no meta-variables) and that sol, sol', etc., are concrete solutions of it. If sol is the solution $\{\!|mol_1, mol_2, \ldots, mol_k|\!\}$ and sol' is the solution $\{\!|mol'_1, mol'_2, \ldots, mol'_l|\!\}$, then $sol \uplus sol'$ is the solution $\{\!|mol_1, mol_2, \ldots, mol_k, mol'_1, mol'_2, \ldots, mol'_l|\!\}$.

1. ***The Reaction Law.*** Given a CHAM rule

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l \quad \in \quad \text{CHAM}$$

   if $mol_1, mol_2, \ldots, mol_k$ and $mol'_1, mol'_2, \ldots, mol'_l$ are (concrete) instances of $m_1\ m_2\ \ldots\ m_k$ and of $m'_1\ m'_2\ \ldots m'_l$ by a common substitution, respectively, then

$$\text{CHAM} \vdash \{\!|mol_1\ mol_2\ \ldots\ mol_k|\!\} \to \{\!|mol'_1\ mol'_2\ \ldots mol'_l|\!\}$$

2. ***The Chemical Law.*** Reactions can be performed freely within any solution:

$$\frac{\text{CHAM} \vdash sol \to sol'}{\text{CHAM} \vdash sol \uplus sol'' \to sol' \uplus sol''}$$

3. ***The Membrane Law.*** A subsolution can evolve freely in any solution context $\{\!|cxt[\Box]|\!\}$:

$$\frac{\text{CHAM} \vdash sol \to sol'}{\text{CHAM} \vdash \{\!|cxt[sol]|\!\} \to \{\!|cxt[sol']|\!\}}$$

4. ***The Airlock Law.***

$$\text{CHAM} \vdash \{\!|mol|\!\} \uplus sol \leftrightarrow \{\!|mol \lhd sol|\!\}$$

Note the unusual fact that $m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$ being a rule in CHAM does not imply that $\text{CHAM} \vdash m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$. Indeed, the CHAM rules are regarded as descriptors of changes that can take place in solutions and only in solutions, while CHAM sequents are incarnations of those otherwise purely abstract rules. What may be confusing is that the same applies also when k and l (the numbers of molecules in the left-hand and right-hand sides of the CHAM rule) happen to be 1 and m_1 and m'_1 happen to be solutions that contain no meta-variables. The two $\to$ arrows, namely the one in CHAM rules and the one in CHAM sequents, ought to be different symbols; however, we adhere to the conventional CHAM notation which uses the same symbol for both. Moreover, when the CHAM is clear from context, we also follow the conventional notation and drop it from sequents, that is, we write $sol \to sol'$ instead of $\text{CHAM} \vdash sol \to sol'$. While we admit that these conventions may sometimes be confusing, in that $sol \to sol'$ can be a rule or a sequent or even both, we hope that the context makes it clear which one is meant.

[8] To avoid inventing new names, it is common to use CHAM both as an abbreviation for "the chemical abstract machine" and as a name of an arbitrary but fixed chemical abstract machine.

The Reaction Law says that CHAM rules can only apply in solutions (wrapped by a membrane), and not arbitrarily wherever they match. The Chemical Law says that once a reaction take place in a certain solution, it can take place in any other larger solution. In other words, the fact that a solution has more molecules than required by the rule does not prohibit the rule from applying. The Reaction and the Chemical laws together say that CHAM rules can apply inside any solutions having some molecules that match the left-had side of the CHAM rule. An interesting case is when the left-hand-side term of the CHAM rule has only one molecule, i.e., when $k = 1$, because the CHAM rule is still allowed to only apply within a solution; it cannot apply in other places where the left-hand side happens to match.

The Membrane Law says that reactions can take place in any solution context. Indeed, $\{\!|cxt[sol]|\!\}$ says that the solution *sol* (which is wrapped in a membrane) appears somewhere, anywhere, inside a solution context $\{\!|cxt[\Box]|\!\}$. Here *cxt* can be any bag-of-molecule context, and we write $cxt[\Box]$ to highlight the fact that it is a context with a hole $\Box$. By wrapping $cxt[\Box]$ in a membrane we enforce a solution context. This rule also suggests that, at any given moment, the global term to rewrite using the CHAM rules should be a solution. Indeed, the CHAM rewriting process gets stuck as soon as the term becomes a proper molecule (not a solution), because the Membrane Law cannot apply.

The Airlock Law is reversible (i.e., it comprises two rewrite rules, one from left-to-right and one from right-to-left) and it allows to extract a molecule from a solution, putting the rest of the solution within a membrane. Using this law one can, for example, rewrite a solution $\{\!|mol_1\ mol_2\ \ldots, mol_k|\!\}$ into $\{\!|mol_1 \lhd \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$. The advantage of doing so is that one can now match the molecule mol_1 within other rules. Indeed, recall that sub-solutions that appear in rules cannot specify any particular molecule term among the rest of the solution, unless the solution contains precisely that molecule. Since $mol_1 \lhd \{\!|mol_2\ \ldots\ mol_k|\!\}$ is a molecule, $\{\!|mol_1 \lhd \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$ can match molecule terms of the form $\{\!|m \lhd s|\!\}$ appearing in CHAM rules, this way one effectively matching (and possibly modifying) the molecule mol_1 via the specific CHAM rules. The Airlock Law is the only means provided by the CHAM to extract or put molecules in a solution.

Note that the four laws above are not claimed to define the CHAM rewriting; they are only properties that the CHAM rewriting should satisfy. In particular, these laws do not capture the concurrency potential of the CHAM.

Definition 19. *The four laws above give us a proof system for CHAM sequents. As usual, if we write* $\text{CHAM} \vdash sol \to sol'$ *in isolation we mean that it is derivable. Also as usual, we let* $\to^*$ *denote the reflexive and transitive closure of* $\to$*, that is,* $\text{CHAM} \vdash sol \to^* sol'$ *if and only if* $sol = sol'$ *or there is some* sol'' *such that* $\text{CHAM} \vdash sol \to sol''$ *and* $\text{CHAM} \vdash sol'' \to^* sol'$*. Finally, we write* $\text{CHAM} \vdash sol \leftrightarrow sol'$ *as a shorthand for the two seqents* $\text{CHAM} \vdash sol \to sol'$ *and* $\text{CHAM} \vdash sol' \to sol$*, and we say that it is derivable if and only if the two sequents are derivable.*

None of the two sequents in Definition 19 captures the underlying concurrent computation of the CHAM. Indeed, $\text{CHAM} \vdash sol \to sol'$ says that one and only one reaction takes place somewhere in *sol*, while $\text{CHAM} \vdash sol \to^* sol'$ says that arbitrarily many steps take place, including ones which can be done concurrently but also ones which can only take place sequentially. Therefore, we can think of the four laws above, and implicitly of the sequents $\text{CHAM} \vdash sol \to sol'$ and $\text{CHAM} \vdash sol \to^* sol'$, as expressing the descriptive capability of the CHAM: what is possible and what is not possible to compute using the CHAM, and not how it operates. We argue that it is still suggestive to think of CHAM reactions as taking place concurrently whenever they do not involve the same molecules, even though this notion is not formalized here. We are actually not aware of any works except this

book that formalize the CHAM concurrency: one can find a representation of CHAM rewriting into K rewriting in Section 9.6, which allows the CHAM to borrow K's concurrent rewriting.

A common source of misunderstanding the CHAM is to wrongly think of CHAM rules as ordinary rewriting rules modulo the associativity, commutativity and identity of the molecule grouping (inside a solution) operation. The major distinction between CHAM rules and such rewrite rules is that the former only apply within solutions (which are wrapped by membranes) no matter whether the rule contains one or more molecules in its left-hand or right-hand terms, while the latter apply anywhere they match. For example, supposing that one extends the syntax of molecules with the syntax of IMP in Section 3.1.1 and one adds a CHAM rule $m + 0 \to m$, then one can rewrite the solution $\{\!|(3 + 0)\ 7|\!\}$ to solution $\{\!|3\ 7|\!\}$, but one cannot rewrite the molecule $5 \,/\, (3 + 0)$ to molecule $5 \,/\, 3$ regardless of what context it is in, because $3 + 0$ is not in a solution. One cannot even rewrite the isolated (i.e., not in a solution context) term $3 + 0$ to 3 in CHAM, for the same reason.

Classification of CHAM Rules

The rules of a CHAM are typically partitioned into three intuitive categories, namely *heating*, *cooling* and *reaction* rules, though there are no formal requirements imposing a rule to be into one category or another. Moreover, the same laws discussed above apply the same way to all categories of rules, and the same restrictions preventing multiset matching apply to all of them.

- *Heating* rules, distinguished by using the relation symbol $\rightharpoonup$ instead of $\to$, are used to structurally rearrange the solution so that reactions can take place.
- *Cooling* rules, distinguished by using the relation symbol $\leftharpoondown$ instead of $\to$, are used after reactions take place to structurally rearrange the solution back into a convenient form, including to remove useless molecules or parts of them.
- *Reaction* rules, which capture the intended computational steps and use the conventional rewrite symbol $\to$, are used to evolve the solution in an irreversible way.

The heating and cooling rules can typically be paired, with each heating rule $l \rightharpoonup r$ having a symmetric cooling rule $l \leftharpoondown r$, so that we can view them as a single bidirectional *heating/cooling* rule. The CHAM notation for writing such heating/cooling rules is the following:

$$l \rightleftharpoons r$$

In particular, it makes sense to regard the airlock axiom as an example of such a heating/cooling bidirectional rule, that is,

$$\{\!|m_1\ m_2\ \ldots\ m_k|\!\} \rightleftharpoons \{\!|m_1 \lhd \{\!|m_2\ \ldots\ m_k|\!\}|\!\}$$

where m_1, m_2, ..., m_k are molecule meta-variables. The intuition here is that we can heat the solution until m_1 is extracted in an airlock, or we can cool it down in which case the airlock m_1 is diffused within the solution. However, we need to assume one such rule for each $k > 0$.

As one may expect, the reaction rules are the heart of the Cham and properly correspond to state transitions. The heating and cooling rules express *structural rearrangements*, so that the reaction rules can match and apply. In other words, we can view the reaction rules as being applied *modulo* the heating and cooling rules. We are going to suggestively use the notation

CHAM $\vdash sol \rightharpoonup sol'$, respectively CHAM $\vdash sol \rightharpoondown sol'$ whenever the rewrite step taking sol to sol' is a heating rule, respectively a cooling rule. Similarly, we may use the notations CHAM $\vdash sol \rightharpoonup^* sol'$ and CHAM $\vdash sol \rightharpoondown^* sol'$ for the corresponding reflexive/transitive closures. Also, to emphasize the fact that there is only one reaction rule applied, we take the freedom to (admittedly ambiguously) write CHAM $\vdash sol \rightarrow sol'$ instead of CHAM $\vdash sol(\rightarrow \cup \rightharpoonup \cup \rightharpoondown)^* sol'$ whenever all the involved rules but one are heating or cooling rules.

3.8.1 The CHAM of IMP

We next show how to give IMP a CHAM semantics. CHAM is particularly well-suited to giving semantics to concurrent distributed calculi and languages, yielding considerably simpler definitions than those afforded by SOS. Since IMP is sequential, it cannot take full advantage of the CHAM's true concurrency capabilities; the multi-threaded IMP++ language discussed in Section 3.5 will make slightly better use of CHAM's capabilities. Nevertheless, some of CHAM's capabilities turn out to be useful even in this sequential language application, others turn out to be deceiving. Our CHAM semantics for IMP below follows in principle the reduction semantics with evaluation contexts definition discussed in Section 3.7.1. One can formally show that a step performed using reduction under evaluation contexts is equivalent to a suite of heating steps, followed by one reaction step, followed by a suite of cooling steps.

The CHAM defined below is just one possible way to give IMP a CHAM semantics. CHAM, like rewriting, is a general framework which does not impose upon its users any particular definitional style. In our case, we chose to conceptually distinguish two types of molecules; we say "conceptually" because, for simplicity, we prefer to define only one *Molecule* syntactic category in our CHAM:

- *Syntactic molecules*, which include all the syntax of IMP in Section 3.1.1, plus all the syntax of its evaluation contexts in Section 3.7.1, plus a mechanism to flatten evaluation contexts; again, for simplicity, we prefer not to include a distinct type of molecule for each distinct syntactic category of IMP.
- *State molecules*, which are pairs $x \mapsto i$, where $x \in Id$ and $i \in Int$.

For clarity, we prefer to keep the syntactic and the state molecules in separate solutions. More precisely, we work with top-level configurations which are solutions of the form

$$\{\!|\{\!|Syntax|\!\}\ \ \{\!|State|\!\}|\!\}$$

Syntax and *State* are solutions containing syntactic and state molecules, respectively. For example,

$$\{\!|\{\!|x := (3 / (x + 2))|\!\}\ \{\!|x \mapsto 1\ \ y \mapsto 0|\!\}|\!\}$$

is a CHAM configuration containing the statement $x := (3 / (x + 2))$ and state $x \mapsto 1, y \mapsto 0$.

The state molecules and implicitly the state solution are straightforward. State molecules are not nested and state solutions are simply multisets of molecules of the form $x \mapsto i$. The CHAM does not allow us to impose constraints on solutions, such as that the molecules inside the state solution indeed define a partial function and not some arbitrary relation (e.g., that there is at most one molecule $x \mapsto i$ for each $x \in Id$). Instead, the state solution will be used in such a way that the original state solution will embed a proper partial function and each rule will preserve this property of it. For example, the CHAM rule for variable assignment, say when assigning integer i to variable x, will match the state molecule as $\{\!|x \mapsto j \lhd \sigma|\!\}$ and will rewrite it to $\{\!|x \mapsto i \lhd \sigma|\!\}$.

The top level syntactic solution holds the current program or fragment of program that is still left to be processed. It is not immediately clear how the syntactic solution should be represented in order to be able to give IMP a CHAM semantics. The challenge here is that the IMP language constructs have evaluation strategies and the subterms that need to be next processed can be arbitrarily deep into the program or fragment of program, such as the framed x in $x := (3 / (\boxed{x} + 2))$. If the CHAM allowed conditional rules, then we could have followed an approach similar to that of SOS described in Section 3.3, reducing the semantics of each language construct to that of its subexpressions or substatements. Similarly, if the CHAM allowed matching using evaluation contexts, then we could have followed a reduction semantics with evaluation contexts approach like the one in Section 3.7.1 which uses only unconditional rules. Unfortunately, the CHAM allows neither conditional rules nor evaluation contexts in matching, so a different approach is needed.

Failed attempts to represent syntax. A natural approach to represent the syntax of a programming language in CHAM may be to try to use CHAM's heating/cooling and molecule/solution nesting mechanisms to decompose syntax unambiguously in such a way that the redex (i.e., the subterm which can be potentially reduced next; see Section 3.7) appears as a molecule in the top syntactic solution. That is, if $p = c[t]$ is a program or fragment of program which can be decomposed in evaluation context c and redex t, then one may attempt to represent it as a solution of the form $\{\!|t\ \gamma_c|\!\}$, where γ_c is some CHAM representation of the evaluation context c. If this worked, then we could use an airlock operation to isolate that redex from the rest of the syntactic solution, i.e. $\{\!|p|\!\} \rightleftharpoons \{\!|t\ \gamma_c|\!\} \rightleftharpoons \{\!|t \lhd \{\!|\gamma_c|\!\}|\!\}$, and thus have it at the same level with the state solution in the configuration solution; this would allow to have rules that match both a syntactic molecule and a state molecule (after an airlock operation is applied on the state solution as well) in the same rule, as needed for the semantics of lookup and assignment. In our example above, we would obtain

$$\{\!|\{\!|x := (3 / (x + 2))|\!\}\ \{\!|x \mapsto 1\ \ y \mapsto 0|\!\}|\!\} \rightleftharpoons \{\!|\{\!|x \lhd \{\!|\gamma_{x := (3 / (\square + 2))}|\!\}|\!\}\ \{\!|x \mapsto 1 \lhd \{\!|y \mapsto 0|\!\}|\!\}|\!\}$$

and the latter could be rewritten with a natural CHAM reaction rule for variable lookup such as

$$\{\!|x \lhd c|\!\}\ \{\!|x \mapsto i \lhd \sigma|\!\} \to \{\!|i \lhd c|\!\}\ \{\!|x \mapsto i \lhd \sigma|\!\}$$

Unfortunately, there seems to be no way to achieve such a desirable CHAM representation of syntax. We next attempt and fail to do it in two different ways, and then give an argument why such a representation is actually impossible.

Consider, again, the statement $x := (3 / (x + 2))$. A naive approach to represent this statement term as a syntactic solution (by means of appropriate heating/cooling rules) is to flatten it into its redex, namely x, and into all its atomic evaluation subcontexts, that is, to represent it as the following solution:

$$\{\!|x\ \ (\square + 2)\ \ (3 / \square)\ \ (x := \square)|\!\}$$

Such a representation can be relatively easily achieved by adding heating/cooling pair rules that correspond to the evaluation strategies (or contexts) of the various language constructs. For example, we can add the following rules corresponding to the evaluation strategies of the assignment and the addition constructs (and two similar ones for the division construct):

$$\begin{array}{rcl} x := a & \rightleftharpoons & a \lhd \{\!|x := \square\ |\!\} \\ a_1 + a_2 & \rightleftharpoons & a_1 \lhd \{\!|\ \square + a_2|\!\} \\ a_1 + a_2 & \rightleftharpoons & a_2 \lhd \{\!|a_1 + \square\ |\!\} \end{array}$$

With such rules, one can now heat or cool syntax as desired, for example:

$$\begin{array}{rcll}
\{\!|x := (3 \,/\, (x+2))|\!\} & \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \lhd \{\!|x := \square\,|\!\}\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \;\; (x := \square)|\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(x+2) \;\; (3 \,/\, \square) \;\; (x := \square)|\!\} & \text{(Reaction, Chemical, Airlock)} \\
& \rightleftharpoons & \{\!|x \;\; (\square+2) \;\; (3 \,/\, \square) \;\; (x := \square)|\!\} & \text{(Reaction, Chemical, Airlock)}
\end{array}$$

Unfortunately this naive approach is ambiguous, because it cannot distinguish the above from the representation of, say, $x := ((3 \,/\, x) + 2)$. The problem here is that the precise structure of the evaluation context is "lost in translation", so the approach above does not work.

Let us attempt a second approach, namely to guarantee that there is precisely one hole $\square$ molecule in each syntactic subsolution by using the molecule/solution nesting mechanism available in CHAM. More precisely, let us try to unambiguously represent the statements $x := (3 \,/\, (x+2))$ and $x := ((3 \,/\, x) + 2)$ as the following two distinct syntactic solutions:

$$\begin{array}{l}
\{\!|x \;\; \{\!|(\square+2) \;\; \{\!|(3 \,/\, \square) \;\; \{\!|(x := \square)|\!\}\!\}\!\}\!\}\!\} \\
\{\!|x \;\; \{\!|(3 \,/\, \square) \;\; \{\!|(\square+2) \;\; \{\!|(x := \square)|\!\}\!\}\!\}\!\}\!\}
\end{array}$$

To achieve this, we modify the heating/cooling rules above as follows:

$$\begin{array}{rcl}
(x := a) \lhd c & \rightleftharpoons & a \lhd \{\!|\{\!|(x := \square) \lhd c|\!\}\!\} \\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_1 \lhd \{\!|\{\!|(\square + a_2) \lhd c|\!\}\!\} \\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_2 \lhd \{\!|\{\!|(a_1 + \square) \lhd c|\!\}\!\}
\end{array}$$

With these modified rules, one may now think that one can heat and cool syntax unambiguously:

$$\begin{array}{rcll}
\{\!|x := (3 \,/\, (x+2))|\!\} & \rightleftharpoons & \{\!|(x := (3 \,/\, (x+2))) \lhd \{\!|\cdot|\!\}\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \lhd \{\!|\{\!|(x := \square) \lhd \{\!|\cdot|\!\}\!\}\!\}\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \lhd \{\!|\{\!|x := \square\,|\!\}\!\}\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|(x+2) \lhd \{\!|\{\!|(3 \,/\, \square) \lhd \{\!|\{\!|x := \square\,|\!\}\!\}\!\}\!\}\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(x+2) \lhd \{\!|\{\!|(3 \,/\, \square) \;\; \{\!|x := \square\,|\!\}\!\}\!\}\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|x \lhd \{\!|\{\!|(\square+2) \lhd \{\!|\{\!|(3 \,/\, \square) \;\; \{\!|x := \square\,|\!\}\!\}\!\}\!\}\!\}\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|x \lhd \{\!|\{\!|(\square+2) \;\; \{\!|(3 \,/\, \square) \;\; \{\!|x := \square\,|\!\}\!\}\!\}\!\}\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|x \;\; \{\!|(\square+2) \;\; \{\!|(3 \,/\, \square) \;\; \{\!|x := \square\,|\!\}\!\}\!\}\!\} & \text{(Airlock)}
\end{array}$$

Unfortunately, the above is not the only way one can heat the solution in question. For example, the following is also a possible derivation, showing that these heating/cooling rules are still problematic:

$$\begin{array}{rcll}
\{\!|x := (3 \,/\, (x+2))|\!\} & \rightleftharpoons & \{\!|(x := (3 \,/\, (x+2))) \lhd \{\!|\cdot|\!\}\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \lhd \{\!|\{\!|(x := \square) \lhd \{\!|\cdot|\!\}\!\}\!\}\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \lhd \{\!|\{\!|x := \square\,|\!\}\!\}\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \;\; \{\!|x := \square\,|\!\}\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(x+2) \;\; \{\!|3 \,/\, \square\,|\!\} \;\; \{\!|x := \square\,|\!\}\!\} & \text{(All four laws)} \\
& \rightleftharpoons & \{\!|x \;\; \{\!|\,\square + 2|\!\} \;\; \{\!|3 \,/\, \square\,|\!\} \;\; \{\!|x := \square\,|\!\}\!\} & \text{(All four laws)}
\end{array}$$

Indeed, one can similarly show that

$$\{\!|x := ((3 \,/\, x) + 2)|\!\} \;\; \rightleftharpoons \;\; \{\!|x \;\; \{\!|\,\square + 2|\!\} \;\; \{\!|3 \,/\, \square\,|\!\} \;\; \{\!|x := \square\,|\!\}\!\}$$

Therefore, this second syntax representation attempt is also ambiguous.

$$
\begin{array}{rcl}
a_1 + a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square + a_2 \curvearrowright c \\
a_1 + a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 + \square \curvearrowright c \\
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,/\, a_2 \curvearrowright c \\
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,/\, \square \curvearrowright c \\
a_1 \texttt{<=} a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \texttt{<=} a_2 \curvearrowright c \\
i_1 \texttt{<=} a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright i_1 \texttt{<=} \square \curvearrowright c \\
\texttt{not}\, b \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{not}\, \square \curvearrowright c \\
b_1 \,\texttt{and}\, b_2 \curvearrowright c & \rightleftharpoons & b_1 \curvearrowright \square \,\texttt{and}\, b_2 \curvearrowright c \\
x \texttt{:=} a \curvearrowright c & \rightleftharpoons & a \curvearrowright x \texttt{:=} \square \curvearrowright c \\
s_1 \,;\, s_2 \curvearrowright c & \rightleftharpoons & s_1 \curvearrowright \square \,;\, s_2 \curvearrowright c \\
s & \rightleftharpoons & s \curvearrowright \square \\
\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{if}\, \square \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c
\end{array}
$$

Figure 3.44: CHAM heating-cooling rules for IMP.

We claim that it is impossible to devise heating/cooling rules in CHAM and representations $\gamma_\_$ of evaluation contexts with the property that

$$\{\!|c[t]|\!\} \rightleftharpoons \{\!|t\ \gamma_c|\!\} \quad \text{or, equivalently,} \quad \{\!|c[t]|\!\} \rightleftharpoons \{\!|t \lhd \{\!|\gamma_c|\!\}|\!\}$$

for any term t and any appropriate evaluation context c. Indeed, if that was possible, then the following derivation could be possible:

$$
\begin{array}{rcll}
\{\!|x \texttt{:=} (3 \,/\, (x + 2))|\!\} & \rightleftharpoons & \{\!|(3 \,/\, (x + 2))\ \ \gamma_{x \texttt{:=} \square}|\!\} & \text{(hypothesis)} \\
& \rightleftharpoons & \{\!|(x + 2)\ \ \gamma_{3 \,/\, \square}\ \ \gamma_{x \texttt{:=} \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x\ \ \gamma_{\square + 2}\ \ \gamma_{3 \,/\, \square}\ \ \gamma_{x \texttt{:=} \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|(3 \,/\, x)\ \ \gamma_{\square + 2}\ \ \gamma_{x \texttt{:=} \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|((3 \,/\, x) + 2)\ \ \gamma_{x \texttt{:=} \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x \texttt{:=} ((3 \,/\, x) + 2)|\!\} & \text{(hypothesis)}
\end{array}
$$

This general impossibility result explains why both our representation attempts above failed, as well as why many other similar attempts are also expected to fail.

The morale of the exercise above is that one should be very careful when using CHAM's airlock, because in combination with the other CHAM laws it can yield unexpected behaviors. In particular, the Chemical Law makes it impossible to state that a term matches the entire contents of a solution molecule, so one should not rely on the fact that all the remaining contents of a solution is in the membrane following the airlock. In our heating/cooling rules above, for example

$$
\begin{array}{rcl}
(x \texttt{:=} a) \lhd c & \rightleftharpoons & a \lhd \{\!|\{\!|(x \texttt{:=} \square) \lhd c|\!\}|\!\} \\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_1 \lhd \{\!|\{\!|(\square + a_2) \lhd c|\!\}|\!\} \\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_2 \lhd \{\!|\{\!|(a_1 + \square) \lhd c|\!\}|\!\}
\end{array}
$$

our intuition that c matches *all* the evaluation context solution representation was wrong precisely for that reason. Indeed, it can just as well match a solution representation of a subcontext, which is why we got the unexpected derivation.

Correct representation of syntax. We next discuss an approach to representing syntax which is *not* based on CHAM's existing solution/membrane mechanism. We borrow from K (see Chapter 5) the idea of flattening syntax in an explicit list of computational tasks. Like in K, we use the symbol $\curvearrowright$, read "then" or "followed by", to separate such computational tasks; to avoid writing parentheses, we here assume that $\curvearrowright$ is right-associative and binds less tightly than any other construct. For example, the term $x := (3 \,/\, (x + 2))$ gets represented as the list term

$$x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square$$

which reads "process x, followed by adding 2 to it, followed by dividing 3 by the result, followed by assigning the obtained result to x, which is the final task". Figure 3.44 shows all the heating/cooling rules that we associate to the various evaluation strategies of the IMP language constructs. These rules allow us to structurally rearrange any well-formed syntactic term so that the next computational task is at the top (left side) of the computation list. The only rule in Figure 3.44 which does not correspond to the evaluation strategy of some evaluation construct is $s \rightleftharpoons s \curvearrowright \square$. Its role is to initiate the decomposition process whenever an unheated statement is detected in the syntax solution. According to CHAM's laws, these rules can only apply in solutions, so we can derive

$$\{\!|x := 1 \,;\, x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^* \{\!|x := 1 \curvearrowright \square \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square|\!\}$$

but there is no way to derive, for example,

$$\{\!|x := 1 \,;\, x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^* \{\!|x := 1 \,;\, (x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square)|\!\}$$

The syntactic solution will contain only one syntactic molecule at any given moment, with no subsolutions, which is the reason why the heating/cooling rules in Figure 3.44 that correspond to language construct evaluation strategies need to mention the remaining of the list of computational tasks in the syntactic molecule, c, instead of just the interesting part (e.g., $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$, etc.), as we do in K (see Section 5). The heating/cooling rules in Figure 3.44 effectively decompose the syntactic term into any of its possible splits into a redex (the top of the resulting list of computational tasks) and an evaluation context (represented flattened as the rest of the list). In fact, these heating/cooling rules have been almost mechanically derived from the syntax of evaluation contexts for the IMP language constructs in Section 3.7.1 (see Figure 3.30).

Figure 3.45 shows the remaining CHAM rules of IMP, giving the actual semantics of each language construct. Like the heating/cooling rules in Figure 3.44, these rules are also almost mechanically derived from the rules of the reduction semantics with evaluation contexts of IMP in Section 3.7.1 (see Figure 3.31), with the following notable differences:

- Each rule needs to mention the remaining list of computational tasks, c, for the same reason the heating/cooling rules in Figure 3.44 need to mention it (which is explained above).
- There is no equivalent of the characteristic rule of reduction semantics with evaluation contexts. The Membrane Law looks somehow similar, but we cannot take advantage of that because we were not able to use the inherent airlock mechanism of the CHAM to represent syntax (see the failed attempts to represent syntax above).
- The state is organized as a solution using CHAM's airlock mechanism, instead of just imported as an external data-structure as we did in our previous semantics. We did so because the state

$$
\begin{array}{rcl}
\{\!|x \rightharpoonup c|\!\}\ \{\!|x \mapsto i \triangleright \sigma|\!\} & \to & \{\!|i \rightharpoonup c|\!\}\ \{\!|x \mapsto i \triangleright \sigma|\!\} \\
i_1 \texttt{ + } i_2 \rightharpoonup c & \to & i_1 +_{Int} i_2 \rightharpoonup c \\
i_1 \texttt{ / } i_2 \rightharpoonup c & \to & i_1 /_{Int} i_2 \rightharpoonup c \quad \text{when } i_2 \neq 0 \\
i_1 \texttt{ <= } i_2 \rightharpoonup c & \to & i_1 \leq_{Int} i_2 \rightharpoonup c \\
\texttt{not true} \rightharpoonup c & \to & \texttt{false} \rightharpoonup c \\
\texttt{not false} \rightharpoonup c & \to & \texttt{true} \rightharpoonup c \\
\texttt{true and } b_2 \rightharpoonup c & \to & b_2 \rightharpoonup c \\
\texttt{false and } b_2 \rightharpoonup c & \to & \texttt{false} \rightharpoonup c \\
\{\!|x \texttt{ := } i \rightharpoonup c|\!\}\ \{\!|x \mapsto j \triangleright \sigma|\!\} & \to & \{\!|\texttt{skip} \rightharpoonup c|\!\}\ \{\!|x \mapsto i \triangleright \sigma|\!\} \\
\texttt{skip ; } s_2 \rightharpoonup c & \to & s_2 \rightharpoonup c \\
\texttt{if true then } s_1 \texttt{ else } s_2 \rightharpoonup c & \to & s_1 \rightharpoonup c \\
\texttt{if false then } s_1 \texttt{ else } s_2 \rightharpoonup c & \to & s_2 \rightharpoonup c \\
\texttt{while } b \texttt{ do } s \rightharpoonup c & \to & \texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else skip} \rightharpoonup c \\
\texttt{var } xl \texttt{ ; } s & \to & \{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\} \\
 & & \\
(x, xl) \mapsto i & \rightharpoonup & x \mapsto i \triangleright \{\!|xl \mapsto i|\!\}
\end{array}
$$

Figure 3.45: CHAM(IMP): The CHAM of IMP, obtained by adding to the heating/cooling rules in Figure 3.44 the semantic rules for IMP plus the heating rule for state initialization above.

data-structure that we used in our previous semantics was a finite-domain partial function (see Section 3.1.2), which was represented as a set of pairs (Section 2.1.2), and it is quite natural to replace any set structures by the inherent CHAM solution mechanism.

Figure 3.46 shows a possible execution of IMP's CHAM defined above, mentioning at each step which of CHAM's laws have been applied. Note that the final configuration contains two subsolutions, one for the syntax and one for the state. Since the syntactic subsolution in the final configuration solution is expected to always contain only `skip`, one can safely eliminated it.

3.8.2 The CHAM of IMP++

We next discuss the CHAM of IMP++, discussing like in the other semantics each feature separately first and then putting all of them together. When putting them together, we also investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The chemical abstract machine can also define the increment modularly:

$$\{\!|\texttt{++}x \rightharpoonup c|\!\}\ \{\!|x \mapsto i \triangleright \sigma|\!\} \to \{\!|i +_{Int} 1 \rightharpoonup c|\!\}\ \{\!|x \mapsto i +_{Int} 1 \triangleright \sigma|\!\} \qquad \text{(CHAM-Inc)}$$

Input/Output

All we have to do is to add new molecules in the top-level solution that hold the input and the output buffers, then define the evaluation strategy of `print` by means of a heating/cooling pair like we did for other strict constructs, and finally to add the reaction rules corresponding to the input output constructs. Since solutions are not typed, to distinguish the solution holding the input

$$
\begin{array}{rcl}
\{\!|\,\texttt{var}\,x,y\,;\,x:=1\,;\,x:=(3\,/\,(x+2))\,|\!\} & \to & \text{(Reaction)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))\,|\!\}\ \ \{\!|x,y\mapsto 0\,|\!\}\,|\!\} & \rightharpoonup & \text{(Heating, Membrane)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x,y\mapsto 0\,|\!\}\,|\!\} & \rightharpoonup^* & \text{(Heating, Membrane, Airlock)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 0\ \ y\mapsto 0\,|\!\}\,|\!\} & \rightharpoonup & \text{(Heating, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\Box\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 0\ \ y\mapsto 0\,|\!\}\,|\!\} & \rightharpoonup & \text{(Airlock, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\Box\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 0\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to & \text{(Reaction)}\\
\{\!|\{\!|\,\texttt{skip}\curvearrowright\Box\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \rightharpoonup & \text{(Cooling, Membrane)}\\
\{\!|\{\!|\,\texttt{skip}\,;\,x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \rightharpoonup & \text{(Reaction, Membrane)}\\
\{\!|\{\!|x:=(3\,/\,(x+2))\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \rightharpoonup^* & \text{(Heating, Membrane)}\\
\{\!|\{\!|x\curvearrowright\Box+2\curvearrowright 3\,/\,\Box\curvearrowright x:=\Box\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to & \text{(Reaction)}\\
\{\!|\{\!|1\curvearrowright\Box+2\curvearrowright 3\,/\,\Box\curvearrowright x:=\Box\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \rightharpoonup & \text{(Cooling)}\\
\{\!|\{\!|1+2\curvearrowright 3\,/\,\Box\curvearrowright x:=\Box\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to & \text{(Reaction, Membrane)}\\
\{\!|\{\!|3\curvearrowright 3\,/\,\Box\curvearrowright x:=\Box\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to^* & \text{(Reaction, Cooling, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to & \text{(Reaction)}\\
\{\!|\{\!|\,\texttt{skip}\curvearrowright\Box\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & \to & \text{(Cooling, Membrane)}\\
\{\!|\{\!|\,\texttt{skip}\,|\!\}\ \ \{\!|x\mapsto 1\triangleright\{\!|y\mapsto 0\,|\!\}\,|\!\}\,|\!\} & &
\end{array}
$$

Figure 3.46: An execution of IMP's CHAM.

buffer from the one holding the output buffer, we introduce two artificial molecules, called input and output, respectively, and place them upfront in their corresponding solutions. Since the input buffer needs to also be provided within the initial solution, we modify the reaction rule for programs to take program and input molecules and initialize the output and state molecule accordingly:

$$
\begin{array}{ll}
\texttt{print(}\,a\,\texttt{)}\curvearrowright c \rightleftharpoons a\curvearrowright\texttt{print(}\,\Box\,\texttt{)}\curvearrowright c & \\
\{\!|\texttt{read()}\curvearrowright c\,|\!\}\ \{\!|\textsf{input}\ i:w\,|\!\}\to\{\!|\texttt{skip}\curvearrowright c\,|\!\}\ \{\!|\textsf{input}\ w\,|\!\} & \text{(CHAM-Read)}\\
\{\!|\texttt{print(}\,i\,\texttt{)}\curvearrowright c\,|\!\}\ \{\!|\textsf{output}\ w\,|\!\}\to\{\!|\texttt{skip}\curvearrowright c\,|\!\}\ \{\!|\textsf{output}\ w:i\,|\!\} & \text{(CHAM-Print)}\\
\{\!|\texttt{var}\,xl\,;\,s\,|\!\}\ \{\!|w\,|\!\}\to\{\!|s\,|\!\}\ \{\!|xl\mapsto 0\,|\!\}\ \{\!|\textsf{input}\ w\,|\!\}\ \{\!|\textsf{output}\ \epsilon\,|\!\} & \text{(CHAM-Pgm)}
\end{array}
$$

Abrupt Termination

The CHAM semantics of abrupt termination is even more elegant and modular than that using evaluation contexts above, because the other components of the configuration need not be mentioned:

$$
i\,/\,0\curvearrowright c\ \ \to\ \ \texttt{skip} \qquad \text{(CHAM-Div-By-Zero)}
$$

$$
\texttt{halt}\curvearrowright c\ \ \to\ \ \texttt{skip} \qquad \text{(CHAM-Halt)}
$$

Dynamic Threads

As stated in Section 3.8, the CHAM has been specifically proposed as a model of concurrent computation, based on the chemical metaphor that molecules in solutions can get together and react, with possibly many reactions taking place concurrently. Since there was no concurrency so far in our language, the actual strength of the CHAM has not been seen yet. Recall that the configuration of the existing CHAM semantics of IMP consists of one top-level solution, which contains two

subsolutions: a syntactic subsolution holding the remainder of the program organized as a molecule sequentializing computation tasks using the special construct $\curvearrowright$; and a state subsolution containing binding molecules, each binding a different program variable to a value. As seen in Figures 3.44 and 3.45, most of the CHAM rules involve only the syntactic molecule. The state subsolution is only mentioned when the language construct involves program variables.

The above suggests that all a `spawn` statement needs to do is to create an additional syntactic subsolution holding the spawned statement, letting the newly created subsolution molecule to float together with the original syntactic molecule in the same top-level solution. Minimalistically, this can be achieved with the following CHAM rule (which does not consider thread termination yet):

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c \,|\!\} \rightarrow \{\!|\, \texttt{skip} \curvearrowright c \,|\!\}\ \{\!|s|\!\}$$

Since the order of molecules in a solution is irrelevant, the newly created syntactic molecule has the same rights as the original molecule in reactions involving the state molecule. We can rightfully think of each syntactic subsolution as an independently running thread. The same CHAM rules we had before (see Figures 3.44 and 3.45) can now also apply to the newly created threads. Moreover, reactions taking place only in the syntactic molecules, which are a majority by a large number, can apply truly concurrently. For example, a thread may execute a loop unrolling step while another thread may concurrently perform an addition. The only restriction regarding concurrency is that rule instances must involve disjoint molecules in order to proceed concurrently. That means that it is also possible for a thread to read or write the state while another thread, truly concurrently, performs a local computation. This degree of concurrency was not possible within the other semantic approaches discussed so far in this chapter.

The rule above only creates threads. It does not collect threads when they complete their computation. One could do that with the simple solution-dissolving rule

$$\{\!|\, \texttt{skip} \,|\!\} \rightarrow \cdot$$

but the problem is that such a rule cannot distinguish between the original thread and the others, so it would also dissolve the original thread when it completes. However, recall that our design decision for each IMP language extension was to always terminate the program normally, no matter whether it uses the new features or not. The IMP normal result configurations contain a syntactic solution and a state solution, the former holding only `skip`. To achieve that, we can flag the newly created threads for collection as below. Here is our complete CHAM semantics of `spawn`:

Molecule ::= ... | `die`

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c \,|\!\} \rightarrow \{\!|\, \texttt{skip} \curvearrowright c \,|\!\}\ \{\!|s \curvearrowright \texttt{die} \,|\!\} \qquad \text{(CHAM-Spawn)}$$

$$\{\!|\, \texttt{skip} \curvearrowright \texttt{die} \,|\!\} \rightarrow \cdot \qquad \text{(CHAM-Die)}$$

We conclude this section with a discussion on the concurrency of the CHAM above. As already argued, it allows for truly concurrent computations to take place, provided that their corresponding CHAM rule instances do not overlap. While this already goes far beyond the other semantical approaches in terms of concurrency, it still enforces interleaving where it should not. Consider, for example, a global configuration in which two threads are about to lookup two different variables in the state cell. Even though there are good reasons to allow the two threads to proceed concurrently, the CHAM above will not, because the two rule instances (of the same CHAM lookup rule) overlap

on the state molecule. This problem can be ameliorated, to some extent, by changing the structure of the top-level configuration to allow all the variable binding molecules currently in the state subsolution to instead float in the top-level solution at the same level with the threads: this way, each thread can independently grab the binding it is interested in without blocking the state anymore. Unfortunately, this still does not completely solve the true concurrency problem, because one could argue that different threads should also be allowed to concurrently read the same variable. Thus, no matter where the binding of that variable is located, the two rule instances cannot proceed concurrently. Moreover, flattening all the syntactic and the semantic ingredients in a top level solution, as the above "fix" suggests, does not scale. Real-life languages can have many configuration items of various kinds, such as, environments, heaps, function/exception/loop stacks, locks held, and so on. Collapsing the contents of all these items in one flat solution would not only go against the CHAM philosophy, but it would also make it hard to understand and control. The K framework (see Section 5) solves this problem by allowing its rules to state which parts of the matched subterm are shared with and which can be concurrently modified by other rules.

Local Variables

The simplest approach to adding blocks with local variables to IMP is to follow and idea similar to the one for reduction semantics with evaluation contexts discussed in Section 3.7.2, assuming the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs:

$$\texttt{let}\, x = a\, \texttt{in}\, s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\, x = \square\, \texttt{in}\, s \curvearrowright c$$
$$\{\!|\texttt{let}\, x = i\, \texttt{in}\, s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \to \{\!|s\ ;\ x := \sigma(x) \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\} \qquad \text{(CHAM-Let)}$$

As it was the case in Section 3.7.2, the above is going to be problematic when we add `spawn` to the language, too. However, recall that in this language experiment we pretend each language extension is final, in order to understand the modularity and flexibility to change of each semantic approach.

The approach above is very syntactic in nature, following the intuitions of evaluation contexts. In some sense, the above worked because we happened to have an assignment statement in our language, which we used for recovering the value of the bound variable. Note, however, that several computational steps were wasted because of the syntactic translations. What we would have really liked to say is "$\texttt{let}\, Id = Int\, \texttt{in}\, Context$ is a special evaluation context where the current state is updated with the binding whenever is passed through top-down, and where the state is recovered whenever is passed through bottom-up". This was not possible to say with evaluation contexts. When using the CHAM, we are free to disobey the syntax. For example, the alternative definition below captures the essence of the problem and wastes no steps:

$$\texttt{let}\, x = a\, \texttt{in}\, s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\, x = \square\, \texttt{in}\, s \curvearrowright c$$
$$\{\!|\texttt{let}\, x = i\, \texttt{in}\, s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoonup \{\!|s \curvearrowright \texttt{let}\, x = \sigma(x)\, \texttt{in}\, \square \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\}$$
$$\{\!|\texttt{skip} \curvearrowright \texttt{let}\, x = v\, \texttt{in}\, \square \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoonup \{\!|\texttt{skip} \curvearrowright c|\!\}\ \{\!|\sigma[v/x]|\!\}$$

In words, the `let` is first heated/cooled in the binding expression. Once that becomes an integer, the `let` is then only heated in its body statement, at the same time updating the state molecule with the binding and storing the return value in the residual `let` construct. Once the `let` body statement becomes `skip`, the solution is cooled down by discarding the residual `let` construct and recovering the state appropriately (we used a "value" v instead of an integer i in the latter rule to indicate the fact that v can also be $\bot$).

Of course, the substitution-based approach discussed in detail in Sections 3.5.6 and 3.7.2 can also be adopted here if one is willing to pay the price for using it:

$$\texttt{let}\ x = a\ \texttt{in}\ s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\ x = \square\ \texttt{in}\ s \curvearrowright c$$
$$\{\!|\texttt{let}\ x = i\ \texttt{in}\ s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \to \{\!|s[x'/x] \curvearrowright c|\!\}\ \{\!|\sigma[i/x']|\!\} \qquad \text{where } x' \text{ is a fresh variable}$$

Putting Them All Together

Putting together all the language features defined in CHAM above is a bit simpler and more modular than in MSOS (see Section 3.6.2): all we have to do is to take the union of all the syntax and semantics of all the features, removing the original rule for the initialization of the solution (that rule was already removed as part of the addition of input/output to IMP); in MSOS, we also had to add the halting attribute to the labels, which we do not have to do in the case of the CHAM.

Unfortunately, like in the case of the reduction semantics with evaluation contexts of IMP++ in Section 3.7.2, the resulting language is flawed. Indeed, a thread spawned from inside a `let` would be created its own molecule in the top-level solution, which would execute concurrently with all the other execution threads, including its parent. Thus, there is the possibility that the parent will advance to the assignment recovering the value of the `let`-bound variable before the spawned thread terminates, in which case the bound variable would be changed in the state by the parent thread, "unexpectedly" for the spawned thread. One way to address this problem is to rename the bound variable into a fresh variable within the `let` body statement, like we did above, using a substitution operation. Another is to split the state into an environment mapping variables to locations and a store mapping locations to values, and to have each thread consist of a solution holding both its code and its environment. Both these solutions were also suggested in Section 3.5.6, when we discussed how to make small-step SOS correctly capture all the behaviors of the resulting IMP++.

3.8.3 CHAM in Rewriting Logic

As explained above, CHAM rewriting cannot be immediately captured as ordinary rewriting modulo solution multiset axioms such as associativity, commutativity and identity. The distinction between the two arises essentially when a CHAM rule involves only one top-level molecule which is not a solution, because the CHAM laws restrict the applications of such a rule only in solutions while ordinary rewriting allows such rules to apply everywhere. To solve this problem, we wrap each rule in a solution context, that is, we translate CHAM rules of the form

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots\ m'_l$$

into corresponding rewriting logic rules of the form

$$\{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \to \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$$

where the only difference between the original CHAM terms m_1, m_2, ..., m_k, m'_1, m'_2, ..., m'_l and their algebraic variants $\overline{m_1}$, $\overline{m_2}$, ..., $\overline{m_k}$, $\overline{m'_1}$, $\overline{m'_2}$, ..., $\overline{m'_l}$ is that the meta-variables appearing in the former (recall that CHAM rules are rule schematas) are turned into variables of corresponding sorts in the latter, and where *Ms* is a variable of sort **Bag{***Molecule***}** that does not appear anywhere else in $\overline{m_1}$, $\overline{m_2}$, ..., $\overline{m_k}$, $\overline{m'_1}$, $\overline{m'_2}$, ..., $\overline{m'_l}$.

With this representation of CHAM rules into rewriting logic rules, it is obvious that rewriting logic's rewriting captures both the Reaction Law and the Chemical Law of the CHAM. What is less

obvious is that it also captures the Membrane Law. Indeed, note that the Membrane Law allows rewrites to take place only when the global term is a solution, while rewriting logic allows rewrites to take place anywhere. However, the rewriting logic rule representation above generates only rules that rewrite solutions into solutions. Thus, if the original term to rewrite is a solution, then so it will stay during the entire rewriting process, and so the Membrane Law is also naturally captured by rewriting logic derivations. However, if the original term to rewrite is a proper molecule (which is not a solution), then so it will stay during the entire rewriting logic's rewriting while the CHAM will not reduce it at all. Still it is important to understand that in this case the corresponding rewriting logic theory can perform rewrite steps (in subsolutions of the original term) which are not possible under the CHAM, in particular that it may lead to non-termination in situations where the CHAM is essentially stuck. To reconcile this inherent difference between the CHAM and rewriting logic, we make the reasonable assumption that the original terms to rewrite can only be solutions. Note that the CHAM sequents in Definition 19 already assume that one only derives solution terms.

The only CHAM law which has not been addressed above is the Airlock Law. Rewriting logic has no builtin construct resembling CHAM's airlock, but its multiset matching is powerful enough to allow us to capture the airlock's behavior through rewrite rules. One possibility is to regard the airlock operation like any other molecular construct. Indeed, from a rewriting logic perspective, the Airlock Law says that any molecule inside a solution can be matched and put into an airlock next to the remaining solution wrapped into a membrane, and this process is reversible. This behavior can be achieved through the following two (opposite) rewrite logic rules, where M is a molecule variable and Ms is a bag-of-molecules variable:

$$\begin{array}{rcl} \{\!|M\ Ms|\!\} & \rightarrow & \{\!|M \triangleright \{\!|Ms|\!\}|\!\} \\ \{\!|M \triangleright \{\!|Ms|\!\}|\!\} & \rightarrow & \{\!|M\ Ms|\!\} \end{array}$$

Another possibility to capture airlock's behavior in rewriting logic is to attempt to eliminate it completely and replace it with matching modulo multiset axioms. While this appears to be possible in many concrete situations, we are however not aware of any general solution to do so systematically for any CHAM. The question is whether the elimination of the airlock is indeed safe, in the sense that the resulting rewriting logic theory does not lose any of the original CHAM's behaviors. One may think that thanks to the restricted form of CHAM's rules, the answer is immediately positive. Indeed, since the CHAM disallows any other constructs for solutions except its builtin membrane operation (a CHAM can only add new syntactic constructs for molecules, but not for solutions) and since solution subterms can either contain only one molecule or otherwise be meta-variables (to avoid multiset matching), we can conclude that in any CHAM rule, a subterm containing an airlock operation at its top can only be of the form $m \triangleright \{\!|m'|\!\}$ or of the form $m \triangleright s$ with s a meta-variable. Both these cases can be uniformly captured as subterms of the form $\overline{m} \triangleright \{\!|ms|\!\}$ with ms a term of sort **Bag{***Molecule***}** in rewriting logic, the former by taking $k = 1$ and $ms = m'$ and the latter by replacing the metavariable s with a term of the form $\{\!|Ms|\!\}$ everywhere in the rule, where Ms is a fresh variable of sort **Bag{***Molecule***}**. Unfortunately, it is not clear how we can eliminate the airlock from subterms of the form $\overline{m} \triangleright \{\!|ms|\!\}$. If such terms appear in a solution or at the top of the rule, then one can replace them by their corresponding bag-of-molecule terms $\overline{m}\ ms$. However, if they appear in a proper molecule context (i.e., not in a solution context), then they cannot be replaced by $\overline{m}\ ms$ (first, we would get a parsing error by placing a bag-of-molecule term in a molecule place; second, reactions cannot take place in ms anymore, because it is not surrounded by a membrane). We cannot replace $\overline{m} \triangleright \{\!|ms|\!\}$ by $\{\!|\overline{m}\ ms|\!\}$ either, because that would cause a double membrane when the thus modified airlock reaches a solution context. Therefore, we keep the airlock.

sorts:
Molecule, *Solution*, **Bag{***Molecule***}**
subsorts:
Solution < *Molecule*
// One may also need to subsort to *Molecule* specific syntactic categories (*Int*, *Bool*, etc.)
operations:
$\{\!|\_|\!\} : \mathbf{Bag\{}Molecule\mathbf{\}} \rightarrow Solution$ // membrane operator
$\_ \triangleright \_ : Molecule \times Solution \rightarrow Molecule$ // airlock operator
// One may also need to define specific syntactic constructs for *Molecule* ($\_ + \_$, $\_ \mapsto \_$, etc.)
rules:
// Add the following two generic (i.e., same for all CHAMs) airlock rewrite logic rules:
$\{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$ // M, Ms variables of sorts *Molecule*, **Bag{***Molecule***}**, resp.
// For each specific CHAM rule $m_1\ m_2\ \ldots m_k \rightarrow m'_1\ m'_2\ \ldots m'_l$ add a rewriting logic rule
$\{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \rightarrow \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$ // Ms variable of sort **Bag{***Molecule***}**
// where $\overline{m}$ replaces each meta-variable in m by a variable of corresponding sort.

Figure 3.47: Embedding of a chemical abstract machine into rewriting logic (CHAM $\rightsquigarrow \mathcal{R}_{\text{CHAM}}$).

Putting all the above together, we can associate a rewriting logic theory to any CHAM as shown in Figure 3.47; for simplicity, we assumed that the CHAM has only one *Molecule* syntactic category and, implicitly, only one corresponding *Solution* syntactic category. In Figure 3.47 and elsewhere in this section, we use the notation *left* $\leftrightarrow$ *right* as a shorthand for two opposite rewrite rules, namely for both *left* $\rightarrow$ *right* and *right* $\rightarrow$ *left*. The discussion above implies the following result:

Theorem 10. ***(Embedding of the chemical abstract machine into rewriting logic)*** *If* CHAM *is a chemical abstract machine, sol and sol′ are two solutions, and* $\mathcal{R}_{\text{CHAM}}$ *is the rewrite logic theory associated to* CHAM *as in Figure 3.47, then the following hold:*

1. CHAM $\vdash sol \rightarrow sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \rightarrow^1 \overline{sol'}$;
2. CHAM $\vdash sol \rightarrow^* sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \rightarrow \overline{sol'}$.

Therefore, one-step solution rewriting in $\mathcal{R}_{\text{CHAM}}$ corresponds precisely to one-step solution rewriting in the original CHAM and thus, one can use $\mathcal{R}_{\text{CHAM}}$ as a replacement for CHAM for any reduction purpose. Unfortunately, this translation of CHAM into rewriting logic does not allow us to borrow the latter's concurrency to obtain the desired concurrent rewriting computational mechanism of the former. Indeed, the desired CHAM concurrency says that "different rule instances can apply concurrently in the same solution as far as they act on different molecules". Unfortunately, the corresponding rewrite logic rule instances cannot apply concurrently according to rewriting logic's semantics because both instances would match the entire solution, including the membrane (and rule instances which overlap cannot proceed concurrently in rewriting logic—see Section 2.7).

The CHAM of IMP in Rewriting Logic

Figure 3.48 shows the rewrite theory $\mathcal{R}_{\text{CHAM(IMP)}}$ obtained by applying the generic transformation procedure in Figure 3.47 to the CHAM of IMP discussed in this section and summarized in

sorts:

$\mathit{Molecule}$, $\mathit{Solution}$, $\mathbf{Bag\{}\mathit{Molecule}\mathbf{\}}$ // generic CHAM sorts

subsorts:

$\mathit{Solution} < \mathit{Molecule}$ // generic CHAM subsort

Int, Bool, $\mathit{Id} < \mathit{Molecule}$ // additional IMP-specific syntactic categories

operations:

$\{\!|\_|\!\} : \mathbf{Bag\{}\mathit{Molecule}\mathbf{\}} \to \mathit{Solution}$ // generic membrane

$\_ \triangleright \_ : \mathit{Molecule} \times \mathit{Solution} \to \mathit{Molecule}$ // generic airlock

$\_ \curvearrowright \_ : \mathit{Molecule} \times \mathit{Molecule} \to \mathit{Molecule}$ // "followed by" operator, for evaluation strategies

// Plus all the IMP language constructs and evaluation contexts (all listed in Figure 3.30),

// collapsing all syntactic categories different from Int, Bool and Id into $\mathit{Molecule}$

rules:

// Airlock:

$\{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$

// Heating/cooling rules corresponding to the evaluation strategies of IMP's constructs:

$\{\!|(A_1 + A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square + A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 + A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 + \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \,/\, A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \,/\, A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \,/\, A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 \,/\, \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright I_1 \texttt{<=} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not}\ B \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{not}\ \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(B_1\ \texttt{and}\ B_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B_1 \curvearrowright \square\ \texttt{and}\ B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(X\ \texttt{:=}\ A \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A \curvearrowright X\ \texttt{:=}\ \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(S_1\ \texttt{;}\ S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(S_1 \curvearrowright \square\ \texttt{;}\ S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|S\ Ms|\!\} \leftrightarrow \{\!|(S \curvearrowright \square)\ Ms|\!\}$

$\{\!|(\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\}$

// Semantic rewrite rules corresponding to reaction computational steps

$\{\!|\{\!|X \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|I \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(I_1 + I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 +_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \,/\, I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 /_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}\ \mathbf{if}\ I_2 \neq 0$

$\{\!|(I_1 \texttt{<=} I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 \leq_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not true} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not false} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{true} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{true and}\ B_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{false and}\ B_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|\{\!|X\ \texttt{:=}\ I \curvearrowright C|\!\}\ \{\!|X \mapsto J \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|\texttt{skip} \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(\texttt{skip ;}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if true then}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_1 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if false then}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{while}\ B\ \texttt{do}\ S \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{if}\ B\ \texttt{then}\ (S\ \texttt{;}\ \texttt{while}\ B\ \texttt{do}\ S)\ \texttt{else skip} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{var}\ xl\ \texttt{;}\ s)\ Ms|\!\} \to \{\!|\{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\}\ Ms|\!\}$

// State initialization:

$\{\!|(X, Xl \mapsto I)\ Ms|\!\} \to \{\!|(X \mapsto I \triangleright \{\!|Xl \mapsto I|\!\})\ Ms|\!\}$

Figure 3.48: $\mathcal{R}_{\mathrm{CHAM(IMP)}}$: The CHAM of IMP in rewriting logic.

Figures 3.44 and 3.45. In addition to the generic CHAM syntax, as indicated in the comment under subsorts in Figure 3.47 we also subsort the builtin sorts of IMP, namely *Int*, *Bool* and *Id*, to *Molecule*. The "followed by" $\_ \curvearrowright \_$ construct for molecules is necessary for defining the evaluation strategies of the various IMP language constructs as explained above; we believe that some similar operator is necessary when defining any programming language whose constructs have evaluation strategies because, as explained above, it appears that the CHAM airlock operator is not suitable for this task. For notational simplicity, we stick to our previous convention that $\_ \curvearrowright \_$ is right associative and binds less tight than any other molecular construct. Finally, we add molecular constructs corresponding to all the syntax that we need in order to define the IMP semantics, which includes syntax for language constructs, for evaluation contexts, and for the state and state initialization.

The rewrite rules in Figure 3.48 are straightforward, following the transformation described in Figure 3.47. We grouped them in four categories: (1) the airlock rule is reversible and precisely captures the Airlock Law of the CHAM; (2) the heating/cooling rules, also reversible, capture the evaluation strategies of IMP's language constructs (see Figure 3.44); (3) the semantic rules are irreversible and capture the computational steps of the IMP semantics (see Figure 3.45); (4) the state initialization rule corresponds to the heating rule in Figure 3.45.

Our CHAM-based rewriting logic semantics of IMP in Figure 3.48 follows blindly the CHAM of IMP in Figures 3.44 and 3.45. All it does is to mechanically apply the transformation in Figure 3.47, without making any attempts to optimize the resulting rewriting logic theory. For example, it is easy to see that the syntactic molecules will always contain only one molecule. Also, it is easy to see that the state molecule can be initialized in such a way that at any moment during the initialization rewriting sequence any subsolution containing a molecule of the form $xl \mapsto i$, with xl a proper list, will contain no other molecule. Finally, one can also notice that the top level solution will always contain only two molecules, namely what we called a syntactic solution and a state solution. All these observations suggest that we can optimize the rewriting logic theory in Figure 3.48 by deleting the variable *Ms* from every rule except the airlock ones. While one can do that for our simple IMP language here, one has to be careful with such optimizations in general. For example, we later on add threads to IMP (see Section 3.5), which implies that the top level solution will contain a dynamic number of syntactic subsolutions, one per thread; if we remove the *Ms* variable from the lookup and assignment rules in Figure 3.48, then those rules will not work whenever there are more than two threads running concurrently. On the other hand, the *Ms* variable in the rule for variable declarations can still be eliminated. The point is that one needs to exercise care when one attempts to hand-optimize the rewrite logic theories resulting from mechanical semantic translations.

☆ The CHAM of IMP in Maude

Like for the previous semantics, it is relatively straightforward to mechanically translate the corresponding rewrite theories into Maude modules. However, unlike in the previous semantics, the resulting Maude modules are not immediately executable. The main problem is that, in spite of its elegant chemical metaphor, the CHAM was not conceived to be blindly executable. For example, most of the heating and cooling rules tend to be reversible, leading to non-termination of the underlying rewrite relation. Non-termination is not a problem per se in rewriting logic and in Maude, because one can still use other formal analysis capabilities of these such as search and model checking, but from a purely pragmatic perspective it is rather inconvenient not to be able to execute an operational semantics of a language, particularly of a simple one like our IMP. Moreover, since the state space of a CHAM can very quickly grow to unmanageable sizes even when the state

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op __ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
  var M : Molecule .  var Ms : Bag{Molecule} .
  rl {| M Ms |} => {| M <| {| Ms |} |} .
  rl {| M <| {| Ms |} |} => {| M Ms |} .
endm

mod IMP-CHAM-SYNTAX is including PL-INT + CHAM .
  subsort Int < Molecule .
--- Define all the IMP constructs as molecule constructs
  op _+_ : Molecule Molecule -> Molecule [prec 33 gather (E e) format (d b o d)] .
  op _/_ : Molecule Molecule -> Molecule [prec 31 gather (E e) format (d b o d)] .
--- ... and so on
--- Add the hole as basic molecular construct, to allow for building contexts as molecules
  op [] : -> Molecule .
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-1 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .
--- + strict in its first argument
  rl {| (A1 + A2) Ms |} => {| (A1 <| {| [] + A2 |}) Ms |} .
  rl {| (A1 <| {| [] + A2 |}) Ms |} => {| (A1 + A2) Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2) Ms |} => {| (A2 <| {| A1 / [] |}) Ms |} .
  rl {| (A2 <| {| A1 / [] |}) Ms |} => {| (A1 / A2) Ms |} .
--- ... and so on
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-2 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .  var C : Solution .
--- + strict in its first argument
  rl {| (A1 + A2 <| C) Ms |} => {| (A1 <| {| {| [] + A2 <| C |} |}) Ms |} .
  rl {| (A1 <| {| {| [] + A2 <| C |} |}) Ms |} => {| (A1 + A2 <| C) Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2 <| C) Ms |} => {| (A2 <| {| {| A1 / [] <| C |} |}) Ms |} .
  rl {| (A2 <| {| {| A1 / [] <| C |} |}) Ms |} => {| (A1 / A2 <| C) Ms |} .
--- ... and so on
endm
```

Figure 3.49: Failed attempts to represent the CHAM of IMP in Maude using the airlock mechanism to define evaluation strategies. This figure also highlights the inconvenience of redefining IMP's syntax (the module `IMP-CHAM-SYNTAX` needs to redefine the IMP syntax as molecule constructs).

space of the represented program is quite small, a direct representation of a CHAM in Maude can easily end up having only a theoretical relevance.

Before addressing the non-termination issue, it is instructive to discuss how a tool like Maude can help us pinpoint and highlight potential problems in our definitions. For example, we have previously seen how we failed, in two different ways, to use CHAM's airlock operator to define the evaluation strategies of the various IMP language constructs. We have noticed those problems with using the airlock for evaluation strategies by actually experimenting with CHAM definitions in Maude, more precisely by using Maude's search command to explore different behaviors of a program. We next discuss how one can use Maude to find out that both our attempts to use airlock for evaluation strategies fail. Figure 3.49 shows all the needed modules. `CHAM` defines the generic syntax of the chemical abstract machine together with its airlock rules in Maude, assuming only one type of molecule. `IMP-CHAM-SYNTAX` defines the syntax of IMP as well as the syntax of IMP's evaluation contexts as a syntax for molecules; since there is only one syntactic category for syntax now, namely `Molecule`, adding $\square$ as a `Molecule` constant allows for `Molecule` to also include all the IMP evaluation contexts, as well as many other garbage terms (e.g., $\square + \square$, etc.). The module `IMP-HEATING-COOLING-CHAM-FAILED-1` represents in Maude, using the general translation of CHAM rules into rewriting logic rules shown in Figure 3.47, heating/cooling rules of the form

$$a_1 + a_2 \quad \rightleftharpoons \quad a_1 \lhd \{\!|\, \square + a_2 \,|\!\}$$

Only two such groups of rules are shown, which is enough to show that we have a problem. Indeed, all four Maude search commands below succeed:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| (3 / 1) + 2 |} .
```

That means that the Maude solution terms `{| 3 / (1 + 2) |}` and `{| (3 / 1) + 2 |}` can rewrite into each other, which is clearly wrong. Similarly, `IMP-HEATING-COOLING-CHAM-FAILED-2` represents in Maude heating/cooling rules of the form

$$(a_1 + a_2) \lhd c \quad \rightleftharpoons \quad a_1 \lhd \{\!|\{\!|(\square + a_2) \lhd c|\!\}|\!\}$$

One can now check that this second approach gives us what we wanted, that is, a chemical representation of syntax where each subsolution directly contains no more than one $\square$:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| ([] + 2) {| 3 / [] |} |} |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| (3 / []) {| [] + 2 |} |} |} .
```

Indeed, both Maude search commands above succeed. Unfortunately, the following commands

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| (3 / 1) + 2 |} .
```

also succeed, showing that our second attempt to use the airlock for evaluation strategies fails, too.

One could also try the following, which should also succeed (proof is the searches above):

```
search[1] {| 3 / (1 + 2) |} =>* {| (3 / 1) + 2 |} .
search[1] {| (3 / 1) + 2 |} =>* {| 3 / (1 + 2) |} .
```

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op __ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
--- The airlock is unnecessary in this particular example.
--- We keep it, though, just in case will be needed as we extend the language.
--- We comment the two airlock rules below out to avoid non-termination.
--- Otherwise we would have to use slower search commants instead of rewrite.
---  var M : Molecule .  var Ms : Bag{Molecule} .
---  rl {| M Ms |} => {| M <| {| Ms |} |} .
---  rl {| M <| {| Ms |} |} => {| M Ms |} .
endm
```

Figure 3.50: Generic representation of the CHAM in Maude.

However, on our machine (Linux, 2.4GHz, 8GB memory) Maude 2 ran out of memory after several minutes when asked to execute any of the two search commands above.

Figures 3.50, 3.51 and 3.52 give a correct Maude representation of CHAM(IMP), based on the rewrite logic theory $\mathcal{R}_{\text{CHAM(IMP)}}$ in Figure 3.48. Three important observations were the guiding factors of our Maude semantics of CHAM(IMP):

1. While the approach to syntax in Figure 3.49 elegantly allows to include the syntax of evaluation contexts into the syntax of molecules by simply defining $\Box$ as a molecular construct, unfortunately it still requires us to redefine the entire syntax of IMP into specific constructs for molecules. This is inconvenient at best. We can do better by simply subsorting to `Molecule` all those syntactic categories that the approach in Figure 3.49 would collapse into `Molecule`. This way, any fragment of IMP code parses to a subsort of `Molecule`, so in particular to `Molecule`. Unfortunately, evaluation contexts are now ill-formed; for example, $\Box$ `+ 2` attempts to sum a molecule with an integer, which does not parse. To fix this, we define a new sort for the $\Box$ constant, say `Hole`, and declare it as a subsort of each IMP syntactic category that is subsorted to `Molecule`. In particular, `Hole` is a subsort of `AExp`, so $\Box$ `+ 2` parses to `AExp`.

2. As discussed above, most of CHAM's heating/cooling rules are reversible and thus, when regarded as rewrite rules, lead to non-termination. To avoid non-termination, we restrict the heating/cooling rules so that heating only applies when the heated subterm has computational contents (i.e., it is not a result) while cooling only applies when the cooled term is completely processed (i.e., it is a result). This way, the heating and the cooling rules are applied in complementary situations, in particular they are not reversible anymore, thus avoiding non-termination. To achieve this, we introduce a subsort `Result` of `Molecule`, together with subsorts of it corresponding to each syntactic category of IMP as well as with an explicit declaration of IMP's results as appropriate terms of corresponding result sort.

3. The CHAM's philosophy is to represent data, in particular program states, using its builtin support for solutions as multisets of molecules, and to use airlock operations to extract pieces of data from such solutions whenever needed. This philosophy is justified both by chemical and by mathematical intuitions, namely that one needs an additional step to observe inside a

```
mod IMP-HEATING-COOLING-CHAM is including IMP-SYNTAX + CHAM .
  sorts Hole ResultAExp ResultBExp ResultStmt Result .
  subsorts Hole < AExp BExp Stmt < Molecule .
  subsorts ResultAExp ResultBExp ResultStmt < Result < Molecule .
  subsorts Int < ResultAExp < AExp .
  subsorts Bool < ResultBExp < BExp .
  subsorts ResultStmt < Stmt .
  op skip : -> ResultStmt [ditto] .

  op [] : -> Hole .
  op _~>_ : Molecule Molecule -> Molecule [gather(e E) prec 120] .

  var X : Id .  var C : [Molecule] .  var A A1 A2 : AExp .  var R R1 R2 : Result .
  var B B1 B2 : BExp .  var I I1 I2 : Int .  var S S1 S2 : Stmt .  var Ms : Bag{Molecule} .

 crl {| (A1 + A2 ~> C) Ms |} => {| (A1 ~> [] + A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] + A2 ~> C) Ms |} => {| (R1 + A2 ~> C) Ms |} .

 crl {| (A1 + A2 ~> C) Ms |} => {| (A2 ~> A1 + [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 + [] ~> C) Ms |} => {| (A1 + R2 ~> C) Ms |} .

 crl {| (A1 / A2 ~> C) Ms |} => {| (A1 ~> [] / A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] / A2 ~> C) Ms |} => {| (R1 / A2 ~> C) Ms |} .

 crl {| (A1 / A2 ~> C) Ms |} => {| (A2 ~> A1 / [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 / [] ~> C) Ms |} => {| (A1 / R2 ~> C) Ms |} .

 crl {| (A1 <= A2 ~> C) Ms |} => {| (A1 ~> [] <= A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] <= A2 ~> C) Ms |} => {| (R1 <= A2 ~> C) Ms |} .

 crl {| (R1 <= A2 ~> C) Ms |} => {| (A2 ~> R1 <= [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> R1 <= [] ~> C) Ms |} => {| (R1 <= R2 ~> C) Ms |} .

 crl {| (not B ~> C) Ms |} => {| (B ~> not [] ~> C) Ms |} if notBool(B :: Result) .
  rl {| (R ~> not [] ~> C) Ms |} => {| (not R ~> C) Ms |} .

 crl {| (B1 and B2 ~> C) Ms |} => {| (B1 ~> [] and B2 ~> C) Ms |} if notBool(B1 :: Result) .
  rl {| (R1 ~> [] and B2 ~> C) Ms |} => {| (R1 and B2 ~> C) Ms |} .

 crl {| (X := A ~> C) Ms |} => {| (A ~> X := [] ~> C) Ms |} if notBool(A :: Result) .
  rl {| (R ~> X := [] ~> C) Ms |} => {| (X := R ~> C) Ms |} .

 crl {| (S1 ; S2 ~> C) Ms |} => {| (S1 ~> [] ; S2 ~> C) Ms |} if notBool(S1 :: Result) .
  rl {| (R1 ~> [] ; S2 ~> C) Ms |} => {| (R1 ; S2 ~> C) Ms |} .

 crl {| S Ms |} => {| (S ~> []) Ms |} if notBool(S :: Result) .
  rl {| (R ~> []) Ms |} => {| R Ms |} .

 crl {| (if B then S1 else S2 ~> C) Ms |} => {| (B ~> if [] then S1 else S2 ~> C) Ms |}
  if notBool(B :: Result) .
  rl {| (R ~> if [] then S1 else S2 ~> C) Ms |} => {| (if R then S1 else S2 ~> C) Ms |} .
endm
```

Figure 3.51: Efficient heating-cooling rules for IMP in Maude.

```
mod IMP-SEMANTICS-CHAM is including IMP-HEATING-COOLING-CHAM + STATE .
  subsort Pgm State < Molecule .
  var X : Id .  var Xl : List{Id} .  var C : Molecule .  var Ms : Bag{Molecule} .
  var Sigma : State .  var B B2 : BExp .  var I J I1 I2 : Int .  var S S1 S2 : Stmt .
  rl {| {| X ~> C |} {| X |-> I & Sigma |} Ms |} => {| {| I ~> C |} {| X |-> I & Sigma |} Ms |} .
  rl {| (I1 + I2 ~> C) Ms |} => {| (I1 +Int I2 ~> C) Ms |} .
 crl {| (I1 / I2 ~> C) Ms |} => {| (I1 /Int I2 ~> C) Ms |} if I2 =/=Bool 0 .
  rl {| (I1 <= I2 ~> C) Ms |} => {| (I1 <=Int I2 ~> C) Ms |} .
  rl {| (not true ~> C) Ms |} => {| (false ~> C) Ms |} .
  rl {| (not false ~> C) Ms |} => {| (true ~> C) Ms |} .
  rl {| (true and B2 ~> C) Ms |} => {| (B2 ~> C) Ms |} .
  rl {| (false and B2 ~> C) Ms |} => {| (false ~> C) Ms |} .
  rl {| {| X := I ~> C |} {| X |-> J & Sigma |} Ms |} => {| {| skip ~> C |} {| X |-> I & Sigma |} Ms |} .
  rl {| (skip ; S2 ~> C) Ms |} => {| (S2 ~> C) Ms |} .
  rl {| (if true  then S1 else S2 ~> C) Ms |} => {| (S1 ~> C) Ms |} .
  rl {| (if false then S1 else S2 ~> C) Ms |} => {| (S2 ~> C) Ms |} .
  rl {| (while B do S ~> C) Ms |} => {| (if B then S ; while B do S else skip ~> C) Ms |} .
  rl {| (var Xl ; S) Ms |} => {| {| S |} {| Xl |-> 0 |} Ms |} .
endm
```

Figure 3.52: The CHAM of IMP in Maude.

solution and, respectively, that multiset matching is a complex operation (it is actually an intractable problem) whose complexity cannot be simply "swept under the carpet". While we agree with these justifications for the airlock operator, one should also note that impressive progress has been made in the last two decades, after the proposal of the chemical abstract machine, in terms of multiset matching. For example, languages like Maude build upon very well-engineered multiset matching techniques. We believe that these recent developments justify us, at least in practical language definitions, to replace the expensive airlock operation of the CHAM with the more available and efficient multiset matching of Maude.

Figure 3.50 shows the generic CHAM syntax that we extend in order to define CHAM(IMP). Since we use Maude's multiset matching instead of state airlock and since we cannot use the airlock for evaluation strategies either, there is effectively no need for the airlock in our Maude definition of CHAM(IMP). Moreover, since the airlock rules are reversible, their introduction would yield non-termination. Consequently, we have plenty of reasons to eliminate them, which is reflected in our `CHAM` module in Figure 3.50. The Maude module in Figure 3.51 defines the evaluation strategies of IMP's constructs and should be now clear: in addition to the syntactic details discussed above, it simply gives the Maude representation of the heating/cooling rules in Figure 3.48. Finally, the Maude module in Figure 3.52 implements the semantic rules and the state initialization heating rule in Figure 3.48, replacing the state airlock operation by multiset matching.

To test the heating/cooling rules, one can write Maude commands such as the two search commands below, asking Maude to search for all heatings of a given syntactic molecule (the sort of X, *Id*, is already declared in the last module):

```
search {| X := 3 / (X + 2) |} =>! Sol:Solution .
search {| X := (Y:Id + Z:Id) / (X + 2) |} =>! Sol:Solution .
```

The former gives only one solution, because heating can only take place on non-result subexpressions

```
Solution 1 (state 4)
states: 5  rewrites: 22 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> 3 / [] ~> X := [] ~> [] |}
```

but the latter gives three solutions:

```
Solution 1 (state 5)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Y:Id ~> [] + Z:Id ~> [] / (X + 2) ~> X := [] ~> [] |}

Solution 2 (state 6)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Z:Id ~> Y:Id + [] ~> [] / (X + 2) ~> X := [] ~> [] |}

Solution 3 (state 7)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> (Y:Id + Z:Id) / [] ~> X := [] ~> [] |}
```

Each of the solutions represents a completely heated term (we used `=>!` in the search commands) and corresponds to a particular order of evaluation of the subexpression in question. The three solutions of the second search command above reflect all possible orders of evaluation allowed by our Maude semantics of CHAM(IMP). Interestingly, note that there is no order in which X is looked up in between Y and Z. This is not a problem for our simple IMP language in this section, but it may result in loss of behaviors when we extend our semantics to IMP++ in Section 3.5. Indeed, it may be that other threads modify the values of X, Y, and Z while the expression above is evaluated by another thread in such a way that behaviors are lost if X is not allowed to be looked up between Y and Z. Consequently, our orientation of the heating/cooling rules came at a price: we lost the fully non-deterministic order of evaluation of the arguments of strict operators; what we obtained is a *non-deterministic choice* evaluation strategy (an order of evaluation is non-deterministically chosen and cannot be changed during the evaluation of the expression—this is discussed in more depth in Section 3.5).

Maude can now act as an execution engine for CHAM(IMP). For example, the Maude command

```
rewrite {| sumPgm |} .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form:

```
rewrites: 7343 in ... cpu (... real) (... rewrites/second)
result Solution: {| {| skip |} {| n |-> 0 & s |-> 5050 |} |}
```

Like in the previous Maude semantics, one can also search for all possible behaviors of a program using `search` commands such as

```
search {| sumPgm |} =>! Sol:Solution .
```

Like before, only one behavior will be discovered (IMP is deterministic so far). However, an unexpectedly large number of states is generated, 3918 versus the 1509 states generated by the previous small-step semantics), mainly due to the multiple ways to apply the heating/cooling rules:

```
Solution 1 (state 3918)
states: 3919  rewrites: 12458 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| {| skip |} {| n |-> 0 & s |-> 5050 |} |}
```

3.8.4 Notes

The chemical abstract machine, abbreviated CHAM, was introduced by Berry and Boudol in 1990 [10, 11]. In spite of its operational feel, the CHAM should not be mistakenly taken for a variant of (small-step) SOS. In fact, Berry and Boudol presented the CHAM as an *alternative* to SOS, to address a number of limitations inherent to SOS, particularly its lack of true concurrency and what they called SOS' "rigidity to syntax". The basic metaphor giving its name to the CHAM was inspired by Banâtre and Le Mètayer's GAMMA language [5, 6, 7], which was the first to view a distributed state as a solution in which many molecules float, and the first to understand concurrent transitions as reactions that can occur simultaneously in many points of the solution. GAMMA was proposed as a highly-parallel programming language, together with a stepwise program derivation approach that allows to develop provably correct GAMMA programs. However, following the stepwise derivation approach to writing GAMMA programs is not as straightforward as writing CHAM rules. Moreover, CHAM's nesting of solutions allows for structurally more elaborate encodings of data and in particular for more computational locality than GAMMA. Also, the CHAM appears to be more suitable as a framework for defining semantics of programming languages than the GAMMA language; the latter was mainly conceived as a programming language itself, suitable for executing parallel programs on parallel machines [4], rather than as a semantic framework.

The distinction between heating, cooling and reaction rules in CHAM is in general left to the user. There are no well-accepted criteria and/or principles stating when a rule should be in one category or another. For example, should a rule that cleans up a solution by removing residue molecules be a heating or a colling rule? We prefer to think of it as a cooling rule, because it falls under the broad category of rules which "structurally rearrange the solution after reactions take place" which we methodologically decided to call cooling rules. However, the authors of the CHAM prefer to consider such rules to be heating rules, with the intuition that "the residue molecules evaporate when heated" [11]. While the distinction between heating and cooling rules may have a flavor of subjectivity, the distinction between actual reaction rules and heating/cooling rules is more important because it gives the computational granularity of one's CHAM. Indeed, it is common to abstract away the heating/cooling steps in a CHAM rewriting sequence as internal steps and then define various relations of interest on the remaining reaction steps possibly relating different CHAMS, such as behavioral equivalence, simulation and/or bisimulation relations [10, 11].

The technique we used in this section to reversibly and possibly non-deterministically sequentialize the program syntax by heating/cooling it into a list of computational tasks was borrowed from the K framework [78, 76, 51, 74] (see Section 5). This mechanism is also reminiscent of Danvy and Nielsen's refocusing technique [22, 23], used to execute reduction semantics with evaluation contexts by decomposing evaluation contexts into stacks and then only incrementally modifying these stacks during the reduction process. Our representation of the CHAM into rewriting logic was inspired from related representations by Șerbănuță *et al.* [85] and by Meseguer[9] [47]. However, our representation differs in that it enforces the CHAM rewriting to only take place in solutions, this way being completely faithful to the intended meaning of the CHAM reactions, while the representations in [85, 47] are slightly more permissive, allowing rewrites to take place everywhere rules match; as explained, this is relevant only when the left-hand side of the rule contains precisely one molecule.

We conclude this section with a note on the concurrency of CHAM rewriting. As already

[9]Meseguer [47] was published in the same volume of the Journal of Theoretical Computer Science as the CHAM extended paper [11] (the first CHAM paper [10] was published two years before, in 1990, same as the first papers on rewriting logic [45, 44, 46])

explained, we are not aware of any formal definition of a CHAM rewriting relation that captures the truly concurrent computation advocated by the CHAM. This is unfortunate, because its concurrency potential is one of the most appealing aspects of the CHAM. Moreover, as already discussed (after Theorem 10), our rewriting logic representation of the CHAM is faithful only with respect to one non-concurrent step and interleaves steps taking place within the same solutions no matter whether they involve common molecules or not, so we cannot borrow rewriting logic's concurrency to obtain a concurrent semantics for the CHAM. What can be done, however, is to attempt a different representation of the CHAM into rewriting logic based on ideas proposed by Meseguer in [48] to capture (a limited form of) graph rewriting by means of equational encodings. The encoding in [48] is theoretically important, but, unfortunately, yields rewrite logic theories which are not feasible in practice using the current implementation of Maude.

3.8.5 Exercises

Exercise 149. *For any CHAM, any molecules mol_1, ..., mol_k, and any $1 \leq i \leq k$, the sequents*

$$\text{CHAM} \vdash \{\!|mol_1 \ \ldots \ mol_k|\!\} \leftrightarrow \{\!|mol_1 \triangleright \{\!|mol_2 \ \ldots mol_i|\!\} \ mol_{i+1} \ldots mol_k|\!\}$$

are derivable, where if $i = 1$ then the bag $mol_2 \ \ldots mol_i$ is the empty bag.

Exercise 150. *Modify the CHAM semantics of* IMP *in Figure 3.45 to use a state data-structure as we did in the previous semantics instead of representing the state as a solution of binding molecules.*

Exercise 151. *Add a cleanup (cooling) rule to the CHAM semantics of* IMP *in Figure 3.45 to remove the useless syntactic subsolution when the computation is terminated. The resulting solution should only contain a state (i.e., it should have the form $\{\!|\sigma|\!\}$, and not $\{\!|\{\!|\sigma|\!\}|\!\}$ or $\{\!|\{\!|\cdot|\!\} \ \{\!|\sigma|\!\}|\!\}$).*

Exercise 152. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that / short-circuits when the numerator evaluates to* 0.
Hint: One may need to make one of the heating/cooling rules for / conditional.

Exercise 153. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that conjunction is non-deterministically strict in both its arguments.*

Exercise 154. *Same as Exercise 66, but for CHAM instead of big-step SOS: add variable increment to* IMP*, like in Section 3.8.2*
☆ *Like for the Maude definition of big-step SOS and unlike for the Maude definitions of small-step SOS, MSOS and reduction semantics with evaluation contexts above, the resulting Maude definition can only exhibit three behaviors (instead of five) of the program* `nondet++Pgm` *in Exercise 66. This limitation is due to our decision (in Section 3.8) to only heat on non-results and cool on results when implementing CHAMs into Maude. This way, the resulting Maude specifications are executable at the expense at losing some of the behaviors due to non-deterministic evaluation strategies.*

Exercise 155. *Same as Exercise 70, but for the CHAM instead of big-step SOS: add input/output to* IMP*, like in Section 3.8.2.*

Exercise 156. *Same as Exercise 74, but for the CHAM instead of big-step SOS: add abrupt termination to* IMP*, like in Section 3.8.2.*

Exercise 157. *Same as Exercise 85, but for the CHAM instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.8.2.*

Exercise* 158. *This exercise asks to define* IMP++ *in CHAM, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the CHAM of* IMP++ *discussed in Section 3.8.2 instead of its small-step SOS in Section 3.5.6.*